**Documentação CltCrossing**

**Npcs:**

Esta classe gerencia os NPCs no jogo, definindo suas animações, inventários, emoções e interações com o jogador.

**Funções da Classe**

**1. func \_ready()**

**Descrição:**

Configura o NPC ao inicializar no jogo.

**Detalhamento do Funcionamento:**

1. **Gera um identificador único (npcId)**:
   * Um objeto da classe RandomNumberGenerator é utilizado para criar um número aleatório entre 0 e 1.000.000.
   * Caso esse número já esteja atribuído a outro NPC no NpcManager, ele gera outro número até que seja único.
2. **Define atributos do NPC**:
   * Associa emotion, inventory\_data e npc\_name aos valores exportados configurados no editor.
3. **Registra o NPC no sistema**:
   * Adiciona o NPC ao NpcManager para ser rastreado e gerenciado.
4. **Atualiza a animação inicial**:
   * Chama a função update\_emotion\_animation para configurar a animação do NPC com base em sua emoção inicial.
5. **Associa missões**:
   * Verifica se o nome do NPC é igual ao que foi pedido. Se for, cria uma nova missão usando o MissionManager e exibe um pop-up notificando o jogador.

**2. func \_physics\_process(delta: float)**

**Descrição:**

Atualiza as animações do NPC a cada frame de simulação física.

**Detalhamento do Funcionamento:**

1. Chama a função play\_animation\_based\_on\_emotion(delta) para aplicar a animação correspondente ao estado emocional atual.

**3. func update\_emotion\_animation()**

**Descrição:**

Atualiza a animação do NPC com base no estado emocional atual.

**Detalhamento do Funcionamento:**

1. Chama play\_animation\_based\_on\_emotion(0) para garantir que a animação esteja sincronizada com o estado emocional.

**4. func play\_animation\_based\_on\_emotion(delta: float)**

**Descrição:**

Reproduz animações baseadas na emoção atual do NPC.

**Detalhamento do Funcionamento:**

1. Usa uma estrutura match para verificar o valor de emotion e reproduzir a animação correspondente através do animation\_player.
2. Para emoções que envolvem movimento (como andar ou correr), ajusta o progresso de path\_follow\_3d multiplicando a velocidade (WALK\_SPEED ou RUN\_SPEED) pelo delta fornecido.

**5. func player\_interact()**

**Descrição:**

Gerencia as ações realizadas ao interagir com o NPC.

**Detalhamento do Funcionamento:**

1. Itera pelos NPCs no NpcManager.
2. Para NPCs específicos (como "Bombeiro"), realiza ações como dropar itens do inventário (drop\_npc\_slot\_data\_by\_name) e carregar o progresso do jogo (SaveLoad.load\_game).

**6. func check\_npc\_items() -> Array**

**Descrição:**

Retorna uma lista de itens presentes no inventário do NPC.

**Detalhamento do Funcionamento:**

1. Cria uma lista vazia chamada items.
2. Itera pelos slots do inventário do NPC usando npc.inventory\_data.slot\_datas.size().
3. Para cada slot que contém um item, obtém o nome do item com get\_slot\_data\_name(i) e o adiciona à lista.
4. Retorna a lista final de itens.

**7. func drop\_item\_from\_npc(npcId: int, index: int)**

**Descrição:**

Remove um item específico do inventário do NPC e o transfere para o jogador.

**Detalhamento do Funcionamento:**

1. Itera pelos NPCs no NpcManager e localiza o NPC correspondente ao npcId.
2. Obtém os dados do slot especificado pelo index.
   * Se o slot não estiver vazio, remove o item e emite o sinal inventory\_updated.
3. Adiciona o item removido ao inventário do jogador chamando PlayerManager.player.inventory\_data.pick\_up\_slot\_data.
4. Exibe um pop-up notificando o jogador sobre o item recebido.

**8. func drop\_all\_npc\_slot\_data()**

**Descrição:**

Remove todos os itens do inventário do NPC.

**Detalhamento do Funcionamento:**

1. Itera pelos slots do inventário do NPC.
2. Chama drop\_item\_from\_npc para cada slot que contém um item.

**9. func drop\_npc\_slot\_data\_by\_name(name: String)**

**Descrição:**

Remove do inventário do NPC todos os itens com o nome especificado.

**Detalhamento do Funcionamento:**

1. Itera pelos slots do inventário.
2. Para cada slot cujo nome corresponde ao parâmetro name, chama drop\_item\_from\_npc para removê-lo.

**10. func remove\_npc\_slot\_data\_by\_index(npcId: int, index: int)**

**Descrição:**

Remove um item de um slot específico do inventário.

**Detalhamento do Funcionamento:**

1. Localiza o NPC correspondente ao npcId.
2. Define o slot do índice especificado como null.
3. Emite o sinal inventory\_updated para atualizar o estado do inventário.

**11. func remove\_npc\_slot\_data\_by\_name(name: String)**

**Descrição:**

Remove itens do inventário do NPC com base no nome.

**Detalhamento do Funcionamento:**

1. Itera pelos slots do inventário.
2. Para cada slot cujo nome corresponde ao parâmetro name, chama remove\_npc\_slot\_data\_by\_index.

**12. func remove\_all\_npc\_slot\_data()**

**Descrição:**

Remove todos os itens do inventário do NPC.

**Detalhamento do Funcionamento:**

1. Itera por todos os slots e chama remove\_npc\_slot\_data\_by\_index para limpá-los.

**13. func get\_npc\_equiped\_slot\_data\_index\_by\_name(name: String) -> int**

**Descrição:**

Retorna o índice do slot que contém o item com o nome especificado.

**Detalhamento do Funcionamento:**

1. Itera pelos slots do inventário do NPC.
2. Se encontrar um slot cujo nome corresponde a name, retorna o índice.
3. Caso contrário, retorna -1.

**Inventário:**

**Classe InventoryData**

Essa classe gerencia os dados do inventário de um jogador, incluindo os slots, a capacidade de adicionar, remover e mesclar itens, além de interagir com a interface do jogo.

**Sinais**

* **inventory\_updated**: Este sinal é emitido sempre que o inventário é alterado (um item é adicionado, removido, ou mesclado). Ele comunica para outros sistemas que a interface do inventário precisa ser atualizada.
* **inventory\_interact**: Emite um sinal quando o jogador interage com um slot do inventário, passando os dados do inventário, o índice do slot e o botão pressionado (por exemplo, clique esquerdo ou direito).

**Funções**

1. **grab\_slot\_data(index: int) -> SlotData**
   * **Descrição**: Essa função é chamada para "agarrar" os dados de um slot de inventário no índice fornecido. Ao fazer isso, ela remove os dados do slot e retorna o item que estava nesse slot.
   * **Passos**:
     + Obtém o SlotData no índice especificado.
     + Se o slot tiver dados, ele é limpo (slot\_datas[index] = null) e o sinal inventory\_updated é emitido.
     + Retorna o SlotData removido, ou null se o slot estava vazio.
2. **drop\_slot\_data(grabbed\_slot\_data: SlotData, index: int) -> SlotData**
   * **Descrição**: Essa função coloca um item no slot especificado. Se o slot já contém um item, a função tenta mesclar os itens se possível. Caso contrário, ela simplesmente coloca o item no slot.
   * **Passos**:
     + Verifica se o slot está vazio.
     + Se o slot já contiver dados, tenta mesclar com o item que foi "agarrado". Se a mesclagem for possível, os itens são combinados.
     + Se os itens não puderem ser mesclados, o item é colocado no slot.
     + Emite o sinal inventory\_updated para atualizar a interface.
     + Retorna o SlotData original, caso seja necessário, ou null se o slot original foi vazio.
3. **drop\_single\_slot\_data(grabbed\_slot\_data: SlotData, index: int) -> SlotData**
   * **Descrição**: Essa função é usada quando o jogador solta um único item de uma pilha de itens. Ela tenta colocar o item em um slot disponível ou cria um novo slot se necessário.
   * **Passos**:
     + Verifica se o slot no índice especificado está vazio.
     + Se o slot estiver vazio, cria um novo SlotData com um único item e coloca nele.
     + Se o slot não estiver vazio, tenta mesclar um único item com o item existente.
     + Se a mesclagem não for possível, percorre o inventário em busca de um slot vazio e coloca o item lá.
     + Emite o sinal inventory\_updated para atualizar a interface.
     + Retorna o grabbed\_slot\_data atualizado, ou null caso a quantidade de itens no slot original tenha sido reduzida a zero.
4. **pick\_up\_slot\_data(slot\_data: SlotData) -> bool**
   * **Descrição**: Essa função tenta pegar um item e adicioná-lo ao inventário. Ela tenta mesclar o item com um slot existente ou adicionar um novo slot, caso não haja espaço.
   * **Passos**:
     + Primeiro, tenta mesclar o item com os slots existentes. Se a mesclagem for bem-sucedida, o item é adicionado ao inventário.
     + Se a mesclagem não for possível, a função procura um slot vazio no inventário e adiciona o item ali.
     + Emite o sinal inventory\_updated para atualizar o inventário na interface.
     + Retorna true se o item foi adicionado ou mesclado, ou false se não houver espaço ou capacidade para o item.
5. **use\_slot\_data(index: int) -> void**
   * **Descrição**: Essa função é usada quando o jogador utiliza um item do inventário (por exemplo, consumindo um item ou utilizando um item de uso único).
   * **Passos**:
     + Obtém os dados do item no slot especificado.
     + Verifica se o item é consumível (uma instância de ItemDataConsumable).
     + Se for consumível, a quantidade do item é reduzida em 1. Se a quantidade atingir 0, o item é removido do slot.
     + Chama a função PlayerManager.use\_slot\_data(slot\_data) para aplicar a ação do item.
     + Emite o sinal inventory\_updated para notificar que o inventário foi alterado.
6. **on\_slot\_clicked(index: int, button: int) -> void**
   * **Descrição**: Essa função é chamada quando um slot do inventário é clicado. Ela emite o sinal inventory\_interact, passando as informações sobre o índice do slot e o botão pressionado.
   * **Passos**:
     + Emite o sinal inventory\_interact com os dados do inventário, o índice do slot e o botão pressionado.
7. **get\_slot\_data\_index\_by\_name(name: String) -> int**
   * **Descrição**: Retorna o índice do slot que possui um item com o nome correspondente.
   * **Passos**:
     + Percorre todos os slot\_datas e verifica se o item possui o nome fornecido.
     + Retorna o índice do slot que contém o item com o nome correspondente.
8. **get\_slot\_data\_description(index: int, description: String) -> String**
   * **Descrição**: Retorna a descrição do item no slot especificado.
   * **Passos**:
     + Obtém os dados do slot no índice especificado.
     + Retorna a descrição do item, ou uma string vazia se o slot estiver vazio.
9. **get\_slot\_data\_name(index: int) -> String**
   * **Descrição**: Retorna o nome do item no slot especificado.
   * **Passos**:
     + Obtém os dados do slot no índice especificado.
     + Retorna o nome do item ou uma string vazia se o slot estiver vazio.
10. **get\_slot\_data\_item(index: int) -> ItemData**
    * **Descrição**: Retorna os dados do item armazenado no slot especificado.
    * **Passos**:
      + Obtém os dados do slot no índice especificado.
      + Retorna os dados do item ou null se o slot estiver vazio.
11. **get\_slot\_data\_quantity(index: int) -> int**
    * **Descrição**: Retorna a quantidade de itens no slot especificado.
    * **Passos**:
      + Obtém os dados do slot no índice especificado.
      + Retorna a quantidade de itens.
12. **can\_pick\_up\_item(item\_data: ItemData) -> bool**
    * **Descrição**: Verifica se um item pode ser adicionado ao inventário.
    * **Passos**:
      + Se o item for único (possui o atributo unique), a função verifica se o item já está no inventário. Se estiver, retorna false (não pode pegar novamente).
      + Caso contrário, retorna true para permitir pegar o item.

**Como o Inventário Funciona**

* A classe InventoryData controla todos os dados relacionados ao inventário do jogador, incluindo a adição, remoção e mesclagem de itens entre os slots. Ela também lida com a interação do jogador com o inventário, disparando sinais sempre que há mudanças nos dados ou quando o jogador interage com um slot.
* Cada slot no inventário contém um SlotData, que contém as informações sobre o item, sua quantidade e outras propriedades específicas (como empilhabilidade).
* As funções são projetadas para fornecer flexibilidade ao gerenciar os itens, permitindo mesclar itens empilháveis, criar novos slots e remover itens conforme necessário.

**Documentação da Classe HotBar**

A classe HotBar representa a barra de atalho do inventário do jogador no jogo, permitindo acesso rápido a até 6 itens selecionados. Ela é herdada de PanelContainer e utiliza uma interface visual para exibir os slots.

**Funções da Classe**

**1. \_unhandled\_key\_input(event: InputEvent) -> void**

**Descrição:**

Lida com entradas de teclas não tratadas para usar itens da barra de atalho.

**Detalhamento do Funcionamento:**

1. **Validações iniciais**:
   * Verifica se a barra de atalho está visível e se o evento é um pressionamento de tecla. Se não, a função retorna sem ação.
2. **Verifica o intervalo de teclas válidas**:
   * Apenas aceita teclas numéricas de 1 a 7 (KEY\_1 a KEY\_7).
3. **Emite o sinal hot\_bar\_use**:
   * Calcula o índice correspondente da tecla pressionada subtraindo KEY\_1 (tecla "1").
   * Emite o índice do slot através do sinal para que o inventário manipule o item associado.

**2. set\_inventory\_data(inventory\_data: InventoryData) -> void**

**Descrição:**

Configura a barra de atalho para refletir os dados do inventário e conecta os sinais necessários.

**Detalhamento do Funcionamento:**

1. **Conecta o sinal inventory\_updated**:
   * Sempre que o inventário for atualizado, a barra será repopulada chamando populate\_hot\_bar.
2. **Preenche a barra inicialmente**:
   * Chama populate\_hot\_bar com os dados do inventário para configurar os slots visuais.
3. **Conecta o sinal de uso**:
   * Liga o sinal hot\_bar\_use ao método use\_slot\_data do inventário, permitindo que itens sejam usados diretamente.

**3. populate\_hot\_bar(inventory\_data: InventoryData) -> void**

**Descrição:**

Atualiza os slots visuais da barra de atalho com os dados do inventário.

**Detalhamento do Funcionamento:**

1. **Remove slots existentes**:
   * Itera por todos os filhos de h\_box\_container (slots atuais) e os remove chamando queue\_free.
2. **Adiciona novos slots**:
   * Itera pelos primeiros 6 slots (slot\_datas.slice(0, 6)) do inventário.
   * Para cada slot\_data:
     + Instancia um novo slot usando Slot.instantiate.
     + Adiciona o slot ao h\_box\_container.
     + Se o slot contém dados (slot\_data não for nulo), chama slot.set\_slot\_data(slot\_data) para associar os dados ao slot.

**Propriedades da Classe**

**1. hot\_bar\_use (Sinal)**

* **Descrição**: Emitido quando o jogador utiliza um item da barra de atalho.
* **Parâmetro**:
  + index: int: Índice do slot utilizado (0 a 6).

**2. Slot (Constante)**

* **Descrição**: Pré-carrega o recurso de cena para um slot individual na barra de atalho.
* **Caminho**: "res://Inventory/Inventory/slot.tscn"

**3. h\_box\_container (Referência)**

* **Descrição**: Gerencia a interface visual dos slots, organizada como um HBoxContainer.
* **Caminho**: $MarginContainer/HBoxContainer

**Fluxo Geral**

1. O jogador pressiona uma tecla (1 a 7).
2. O evento é capturado por \_unhandled\_key\_input, e o índice do slot é emitido via hot\_bar\_use.
3. O inventário manipula o uso do item correspondente ao índice.
4. Quando o inventário é atualizado, populate\_hot\_bar reorganiza os slots exibidos na barra de atalho.

**Documentação da Classe InventoryInterface**

A classe InventoryInterface gerencia a interface gráfica do inventário do jogador, exibindo os itens em um formato de grade. É herdada de PanelContainer e organiza slots para representar os itens do inventário.

**Funções da Classe**

**1. set\_inventory\_data(inventory\_data: InventoryData) -> void**

**Descrição:**

Configura os dados do inventário e conecta os sinais necessários para atualizar a interface gráfica.

**Detalhamento do Funcionamento:**

1. **Conexão do sinal inventory\_updated**:
   * Liga o evento de atualização do inventário (inventory\_updated) à função populate\_item\_grid. Isso garante que sempre que o inventário for modificado, a interface seja atualizada automaticamente.
2. **Popula a grade com os itens**:
   * Chama populate\_item\_grid para carregar e exibir os dados atuais do inventário.

**2. clear\_inventory\_data(inventory\_data: InventoryData) -> void**

**Descrição:**

Remove a configuração atual do inventário e desconecta os sinais.

**Detalhamento do Funcionamento:**

1. **Desconexão do sinal inventory\_updated**:
   * Remove a ligação entre o evento de atualização do inventário e a função populate\_item\_grid. Isso impede atualizações indesejadas na interface quando o inventário é alterado após a limpeza.

**3. populate\_item\_grid(inventory\_data: InventoryData) -> void**

**Descrição:**

Atualiza a grade de slots com os dados fornecidos pelo inventário.

**Detalhamento do Funcionamento:**

1. **Limpa a grade existente**:
   * Itera por todos os filhos do nó item\_grid e os remove usando queue\_free. Isso garante que nenhuma interface residual permaneça.
2. **Adiciona novos slots**:
   * Para cada SlotData em inventory\_data.slot\_datas:
     + Instancia um novo slot usando Slot.instantiate().
     + Adiciona o slot à grade (item\_grid).
3. **Conecta sinais de clique**:
   * Liga o sinal slot\_clicked de cada slot ao método on\_slot\_clicked do inventário. Isso permite que cliques nos slots interajam diretamente com os dados do inventário.
4. **Configura dados no slot**:
   * Caso o slot tenha dados associados (slot\_data não seja nulo), o método slot.set\_slot\_data(slot\_data) é chamado para exibir os dados no slot.

**Propriedades da Classe**

**1. Slot (Constante)**

* **Descrição**: Pré-carrega o recurso da cena que representa um slot individual no inventário.
* **Caminho**: "res://Inventory/Inventory/slot.tscn"

**2. item\_grid (Referência)**

* **Descrição**: Nó do tipo GridContainer que organiza os slots de itens em uma grade.
* **Caminho**: $MarginContainer/ItemGrid

**Fluxo Geral**

1. O método set\_inventory\_data é chamado para associar os dados do inventário à interface.
2. A função populate\_item\_grid cria e organiza os slots com base nos itens atuais do inventário.
3. Quando o inventário é atualizado, o sinal inventory\_updated chama novamente populate\_item\_grid, garantindo que a interface reflita as mudanças.
4. O método clear\_inventory\_data pode ser usado para desconectar os sinais e limpar os slots quando a interface do inventário não for mais necessária.

**Documentação da Classe SlotData**

A classe SlotData representa um slot de inventário no sistema do jogo. Cada slot contém informações sobre o item armazenado e sua quantidade. É uma classe fundamental para gerenciar a lógica de itens em um inventário, como empilhamento, fusão de slots e criação de novos slots.

**Propriedades da Classe**

**1. MAX\_STACK\_SIZE (Constante)**

* **Descrição**: Define o número máximo de itens que podem ser empilhados em um único slot.
* **Valor padrão**: 99.

**2. item\_data (Variável Exportada)**

* **Descrição**: Contém os dados do item associado ao slot.
* **Tipo**: ItemData.

**3. quantity (Variável Exportada)**

* **Descrição**: Indica a quantidade de itens no slot.
* **Tipo**: Inteiro (valor mínimo: 1, máximo: MAX\_STACK\_SIZE).
* **Validação adicional**: Se o item não for empilhável, a quantidade máxima é automaticamente limitada a 1.

**Funções da Classe**

**1. can\_merge\_with(other\_slot\_data: SlotData) -> bool**

**Descrição:**

Verifica se dois slots podem ser mesclados parcialmente.

**Como funciona:**

* Confere as seguintes condições:
  1. Os itens nos dois slots são do mesmo tipo (item\_data == other\_slot\_data.item\_data).
  2. Os itens são empilháveis (item\_data.stackable).
  3. A quantidade atual no slot é menor que o limite máximo (quantity < MAX\_STACK\_SIZE).
* Retorna true se todas as condições forem satisfeitas; caso contrário, retorna false.

**2. can\_fully\_merge\_with(other\_slot\_data: SlotData) -> bool**

**Descrição:**

Verifica se dois slots podem ser mesclados completamente.

**Como funciona:**

* Similar a can\_merge\_with, mas adiciona uma condição adicional:
  + A soma das quantidades dos dois slots não excede o limite máximo (quantity + other\_slot\_data.quantity <= MAX\_STACK\_SIZE).

**3. create\_single\_slot\_data() -> SlotData**

**Descrição:**

Cria um novo slot com apenas um item do slot atual e reduz a quantidade no slot original.

**Como funciona:**

1. Duplica o slot atual usando duplicate().
2. Define a quantidade do novo slot como 1.
3. Reduz a quantidade no slot original em 1.
4. Retorna o novo slot criado.

**4. fully\_merge\_with(other\_slot\_data: SlotData) -> void**

**Descrição:**

Mescla completamente o conteúdo de outro slot com o slot atual.

**Como funciona:**

* Adiciona a quantidade do outro slot (other\_slot\_data.quantity) à quantidade do slot atual (quantity).

**5. set\_quantity(value: int) -> void**

**Descrição:**

Define a quantidade de itens no slot, garantindo que os itens não empilháveis tenham no máximo uma unidade.

**Como funciona:**

1. Define o valor da quantidade para o slot.
2. Verifica se a quantidade excede 1 e se o item não é empilhável (!item\_data.stackable).
   * Caso verdadeiro, ajusta a quantidade para 1.

Vamos detalhar a classe que gerencia o comportamento de um **slot de inventário** na interface, que herda de PanelContainer. Ela lida com a exibição e interação com o slot do inventário, mostrando a textura do item, a quantidade de itens e respondendo a eventos de clique do mouse.

**Classe Slot (herdando de PanelContainer)**

Essa classe gerencia a exibição de um único slot de inventário, controlando a interface gráfica do item no slot (textura, quantidade) e respondendo aos cliques do jogador.

**Sinal**

* **slot\_clicked(index: int, button: int)**: Esse sinal é emitido quando o jogador clica em um slot. O sinal passa o índice do slot e o botão do mouse (esquerdo ou direito) que foi pressionado. Esse sinal permite que a lógica do inventário ou outras classes se comuniquem com o slot clicado.

**Propriedades**

* **@onready var texture\_rect: TextureRect = $MarginContainer/TextureRect**:
  + Referência ao TextureRect dentro do PanelContainer que exibe a textura do item no slot. Essa propriedade permite que o código altere a aparência visual do item no slot.
* **@onready var quantity\_label: Label = $QuantityLabel**:
  + Referência ao Label dentro do slot que exibe a quantidade de itens. Esse rótulo é mostrado ou ocultado dependendo da quantidade do item no slot.

**Funções**

1. **\_ready() -> void**
   * **Descrição**: Essa função é chamada quando o nó (PanelContainer) está pronto, ou seja, quando a interface gráfica foi carregada.
   * **Passos**:
     + Define a propriedade mouse\_filter como Control.MOUSE\_FILTER\_PASS, permitindo que o slot detecte eventos de entrada do mouse (como cliques).
2. **set\_slot\_data(slot\_data: SlotData) -> void**
   * **Descrição**: Atualiza a aparência e os dados do slot, incluindo a textura do item, o texto do tooltip e a quantidade de itens.
   * **Passos**:
     + Obtém os dados do item (item\_data) a partir de slot\_data.
     + **Se item\_data não for null**:
       - Define a textura do item (texture\_rect.texture = item\_data.texture).
       - Atualiza o texto do tooltip\_text com o nome, a descrição e o preço do item. O tooltip é o texto que aparece ao passar o mouse sobre o slot.
       - Se a quantidade de itens for maior ou igual a 1, exibe a quantidade de itens no quantity\_label. Caso contrário, o quantity\_label é escondido.
     + **Se item\_data for null**:
       - Limpa a textura do slot e o texto do tooltip.
       - Esconde o rótulo de quantidade.
3. **\_on\_gui\_input(event: InputEvent) -> void**
   * **Descrição**: Essa função responde aos eventos de entrada no slot, como cliques de mouse.
   * **Passos**:
     + Verifica se o evento é um clique do mouse (botão esquerdo ou direito).
     + Se o evento for um clique de mouse e o botão estiver pressionado, emite o sinal slot\_clicked, passando o índice do slot e o botão pressionado (esquerdo ou direito).

**Interação com o Slot**

* **Atualizando o Slot**: A função set\_slot\_data permite atualizar o slot com novos dados de item, alterando a textura do item e o rótulo de quantidade. Isso é útil para refletir as mudanças no inventário, como quando um item é adicionado ou removido.
* **Detectando Cliques**: A função \_on\_gui\_input detecta cliques nos slots. Quando um jogador clica no slot (com o botão esquerdo ou direito), o sinal slot\_clicked é emitido. Esse sinal pode ser utilizado para acionar outras ações, como abrir o menu de uso de item, ou comprar e vender itens, dependendo da implementação do sistema.

**Classe Principal que Herda de Node**

Essa classe principal gerencia a interface do inventário do jogador, a interação com itens e a alternância de visibilidade da interface do inventário.

**Variáveis e Nós Referenciados**

1. **const PickUp = preload("res://Inventory/Item/PickUp/pick\_up.tscn")**
   * **Descrição**: Carrega o arquivo de cena para o item que pode ser pego. Esse arquivo de cena será instanciado quando um item for solto na interface do inventário, criando a representação do item no mundo.
2. **@onready var player: CharacterBody3D = $Player**
   * **Descrição**: Referência ao nó do jogador (CharacterBody3D), permitindo acessar a lógica relacionada ao jogador, como seu inventário e ações.
3. **@onready var inventory\_interface: Control = $Ui/InventoryInterface**
   * **Descrição**: Referência ao nó de interface do inventário (InventoryInterface). Essa interface é usada para exibir e interagir com o inventário do jogador.
4. **@onready var hot\_bar\_inventory: PanelContainer = $Ui/HotBarInventory**
   * **Descrição**: Referência à barra de inventário rápido (HotBarInventory), que pode ser usada para acessar itens rapidamente, sem abrir o inventário completo.

**Funções**

1. **\_ready() -> void**
   * **Descrição**: Função executada quando o nó (a classe principal) é carregado e pronto para começar a ser usado na cena.
   * **Passos**:
     + Conecta o sinal de alternância do inventário do jogador à função toggle\_inventory\_interface.
     + Define os dados do inventário do jogador na interface de inventário.
     + Define os dados de inventário de equipamentos na interface.
     + Conecta a função de alternância do inventário para todos os inventários externos no grupo external\_inventory. Isso permite que outros inventários fora do jogador também possam alternar a visibilidade da interface de inventário.
     + Chama a função SaveLoad.load\_game() para carregar o progresso salvo do jogo, se necessário.
2. **toggle\_inventory\_interface(external\_inventory\_owner = null) -> void**
   * **Descrição**: Função responsável por alternar a visibilidade da interface do inventário.
   * **Passos**:
     + Alterna a visibilidade da interface do inventário (inventory\_interface.visible).
     + Se a interface do inventário estiver visível, mostra o cursor do mouse, caso contrário, captura o cursor para o jogo.
     + Se um external\_inventory\_owner for fornecido e a interface do inventário estiver visível, configura a interface para mostrar o inventário do proprietário externo. Caso contrário, limpa o inventário externo.
3. **\_on\_inventory\_interface\_drop\_slot\_data(slot\_data: SlotData) -> void**
   * **Descrição**: Função que é chamada quando um item é solto na interface do inventário.
   * **Passos**:
     + Instancia um novo item que pode ser pego (PickUp.instantiate()).
     + Define os dados do slot do item para o item a ser pego.
     + Define a posição do item no mundo com base na posição de drop do jogador.
     + Adiciona o item à árvore de nós (add\_child(pick\_up)), tornando-o visível e interativo no mundo do jogo.

**Funcionamento Geral**

* **Conexão de Sinais e Funções**:
  + A função toggle\_inventory\_interface é chamada sempre que o jogador (ou qualquer outro inventário externo) alterna a visibilidade do inventário. Isso altera a interface do inventário e também o comportamento do cursor do mouse, que pode ser capturado ou visível dependendo da visibilidade da interface.
* **Gerenciamento de Itens**:
  + Quando um item é solto no inventário, a função \_on\_inventory\_interface\_drop\_slot\_data é chamada. Ela cria uma instância do item a ser pego, posiciona o item no mundo e adiciona esse item como um novo nó, permitindo que ele seja pegável e interaja com o ambiente do jogo.
* **Integração com o Inventário Externo**:
  + Se o jogador interagir com um inventário externo (como um vendedor ou baú), a função toggle\_inventory\_interface pode ser chamada com um external\_inventory\_owner, o que atualiza a interface para exibir os itens desse inventário específico.

**Classe chest para inventários externos:**

**Variáveis e Nós Referenciados**

1. **@export var inventory\_data: InventoryData**
   * **Descrição**: Esta variável exportada contém os dados do inventário do baú, utilizando a classe InventoryData. Os itens do baú serão armazenados aqui e podem ser acessados ou modificados pelo jogador.
2. **@onready var animation\_player: AnimationPlayer = $chest2/AnimationPlayer**
   * **Descrição**: Referência ao AnimationPlayer dentro do baú, que gerencia as animações de abertura e fechamento do baú.
3. **@onready var external\_inventory\_label: Label = $"../../Ui/InventoryInterface/ExternalInventory/ExternalInventoryLabel"**
   * **Descrição**: Referência ao rótulo (label) da interface do usuário que exibe o nome do tipo do inventário do baú (exemplo: "Baú de Tesouro").
4. **@onready var external\_inventory\_node: Node2D = $"../../Ui/InventoryInterface/ExternalInventory"**
   * **Descrição**: Referência ao nó que controla a visibilidade do inventário externo do baú na interface do usuário.
5. **@export var name\_type: String**
   * **Descrição**: Variável exportada que define o tipo do baú, usado para exibir o nome do baú na interface quando o jogador interage com ele (exemplo: "Baú", "Armazenamento", etc.).

**Funções**

1. **\_ready() -> void**
   * **Descrição**: Função que é chamada quando o baú é carregado na cena.
   * **Passos**:
     + Define o baú como o inventário externo atual, atribuindo-o à variável external\_inventory do ExternalInventoryManager.
     + Reproduz a animação de "fechado" assim que o baú é carregado.
2. **player\_interact() -> void**
   * **Descrição**: Função chamada quando o jogador interage com o baú (geralmente ao clicar nele).
   * **Passos**:
     + Emite o sinal toggle\_inventory, passando a instância do baú. Isso serve para alternar a visibilidade do inventário do baú.
     + Atualiza o texto do rótulo external\_inventory\_label para o nome do tipo do baú (exemplo: "Baú de Tesouro").
     + Se o inventário estiver visível (external\_inventory\_node.visible), o baú reproduz a animação de "aberto". Caso contrário, reproduz a animação de "fechado".
3. **external\_inventory\_have\_this\_item(item\_data: ItemData) -> bool**
   * **Descrição**: Função que verifica se o baú contém um item específico no seu inventário.
   * **Passos**:
     + Itera sobre os slots do inventário do baú (acessando ExternalInventoryManager.external\_inventory.inventory\_data.slot\_datas).
     + Para cada slot, verifica se o item armazenado no slot tem o mesmo nome do item que está sendo procurado (item\_data.name) e se o item é único (item\_data.unique).
     + Se o item for encontrado, a função retorna true. Caso contrário, retorna false.

**Funcionamento Geral**

* **Interação com o Baú**:
  + Quando o jogador interage com o baú, a função player\_interact é chamada. Isso alterna a visibilidade do inventário do baú e executa as animações de abrir ou fechar o baú.
  + O rótulo de interface é atualizado para exibir o tipo do baú, permitindo que o jogador saiba com qual baú ele está interagindo (exemplo: "Baú de Tesouro").
* **Verificação de Itens no Inventário**:
  + A função external\_inventory\_have\_this\_item permite que o jogo verifique se o baú já contém um item específico, considerando a propriedade de item único (unique). Isso é útil para evitar que itens duplicados sejam colocados no baú se eles forem definidos como exclusivos.
* **Interface Externa**:
  + A interface externa do baú permite que o jogador acesse o inventário do baú para adicionar ou retirar itens. O controle de visibilidade e a animação de abrir/fechar melhoram a experiência de interação com o baú no jogo.

**Classe pick-up:**

Claro! Vou detalhar o comportamento da classe PickUp de forma organizada e visualmente clara, para que você possa entender as funcionalidades de forma estruturada, como eu estava fazendo antes:

**1. Variáveis e Referências**

* **@export var slot\_data: SlotData**
  + **Tipo**: SlotData
  + **Função**: Armazena os dados do item que o jogador pode pegar, incluindo informações como a textura, nome, quantidade, preço, etc. O SlotData é um tipo de container que agrupa as informações sobre o item, enquanto o item\_data dentro dele fornece acesso aos atributos específicos do item.
* **@onready var sprite\_3d: Sprite3D = $Sprite3D**
  + **Tipo**: Sprite3D
  + **Função**: Representa a imagem 3D do item no mundo do jogo. O Sprite3D é usado para exibir o item, e a textura é atribuída através de slot\_data.item\_data.texture. Esse nó se move e gira conforme o item interage com o mundo.

**2. Funções**

* **\_ready()**
  + **Descrição**: Esta função é chamada quando a instância do item (PickUp) é criada e adicionada à cena.
  + **Funcionalidade**:
    - A função acessa os dados do item através de slot\_data.item\_data e define a textura do sprite\_3d para corresponder à textura do item. Isso garante que o item visível no mundo tenha a aparência do item correspondente no inventário do jogador.
* **\_physics\_process(delta: float)**
  + **Descrição**: Esta função é chamada a cada quadro, durante o processo físico do jogo. Ela é chamada continuamente e é usada para manipular a física do objeto.
  + **Funcionalidade**:
    - A função aplica uma rotação constante no eixo Y do sprite\_3d usando rotate\_y(delta). Isso cria um efeito visual de rotação contínua, fazendo com que o item gire suavemente enquanto está presente no mundo, atraindo a atenção do jogador.
* **\_on\_area\_3d\_body\_entered(body: Node3D)**
  + **Descrição**: Esta função é chamada quando outro corpo (como o jogador) entra na área de colisão do PickUp. O PickUp geralmente está envolto em uma área de detecção (como uma Area3D) para saber quando o jogador interage com ele.
  + **Funcionalidade**:
    - A função verifica se o corpo que entrou na área de colisão pertence ao jogador, verificando se o body está no grupo "Player". Isso é feito usando body.is\_in\_group("Player").
    - Caso o jogador tenha entrado em contato com o item, a função tenta adicionar o item ao inventário do jogador. A função body.inventory\_data.pick\_up\_slot\_data(slot\_data) é usada para verificar se o item pode ser pegado e, se possível, adicioná-lo ao inventário.
    - Se o item for adicionado ao inventário do jogador, a função queue\_free() é chamada para remover o PickUp da cena e liberar seus recursos, fazendo com que o item desapareça do mundo.

**3. Interatividade com o Jogador**

* **Entrada do Jogador na Área de Colisão**:
  + O item é interativo no mundo 3D. Quando o jogador entra na área de colisão, o item é detectado, e a possibilidade de pegar o item é acionada.
  + O código garante que apenas o jogador possa pegar o item, utilizando o grupo "Player", tornando-o um comportamento específico para o personagem do jogador.
* **Adição ao Inventário**:
  + Ao pegar o item, ele é transferido para o inventário do jogador por meio da função pick\_up\_slot\_data(slot\_data), que é responsável por adicionar a instância do item ao inventário de forma controlada, levando em consideração as regras do jogo (como a quantidade máxima de itens ou se o item é único).

**4. Resumo do Fluxo de Funcionalidade**

* O item aparece no mundo 3D com a textura correta atribuída através dos dados do inventário.
* Ele gira suavemente no mundo para atrair a atenção do jogador.
* Quando o jogador entra em contato com o item, o sistema verifica se ele pode pegá-lo.
* Se for possível, o item é adicionado ao inventário do jogador e desaparece do mundo.

**Classe mission:**

**1. Variáveis e Referências**

* **@export var title: String**
  + **Tipo**: String
  + **Função**: Armazena o título da missão, o nome que será exibido ao jogador. Isso ajuda a identificar a missão no jogo.
* **@export var description: String**
  + **Tipo**: String
  + **Função**: Armazena a descrição detalhada da missão. Esta descrição explica o objetivo da missão para o jogador.
* **@export var npc\_name: String**
  + **Tipo**: String
  + **Função**: Armazena o nome do NPC associado à missão. Pode ser utilizado para verificar a emoção do NPC necessária para completar a missão, ou para fornecer detalhes adicionais sobre qual NPC o jogador deve interagir.
* **@export var is\_completed: bool = false**
  + **Tipo**: bool
  + **Função**: Define o estado da missão. Se true, a missão foi completada. Caso contrário, o jogador ainda precisa completá-la. Inicialmente é definida como false, indicando que a missão começa como não concluída.
* **@export var reward: String**
  + **Tipo**: String
  + **Função**: Define a recompensa pela missão. A recompensa pode ser um item, uma quantidade de experiência ou outro tipo de benefício que o jogador receberá ao completar a missão.

**2. Funções e Métodos**

**check\_completion()**

* **Descrição**: Verifica se a missão foi completada.
* **Função**:
  + Se o estado de is\_completed for true, significa que a missão foi concluída, então a função retorna true.
  + Se o estado for false, a função retorna false, indicando que a missão ainda não foi completada.

**complete\_mission(mission: Mission)**

* **Descrição**: Marca a missão como completada e concede a recompensa ao jogador.
* **Função**:
  + A função verifica se a missão ainda não foi completada (if !mission.is\_completed).
  + Se a missão não foi completada, ela é adicionada à lista de missões completadas do jogador (PlayerManager.player.missions\_complete.append(mission)).
  + Depois disso, o estado da missão é atualizado para true, e uma mensagem é impressa no console indicando que a missão foi completada.
  + A função chama o método give\_reward() para fornecer a recompensa ao jogador.

**give\_reward()**

* **Descrição**: Concede a recompensa ao jogador, seja um item, experiência ou outra forma de recompensa.
* **Função**:
  + A função imprime no console a recompensa concedida (isso pode ser um item ou uma quantidade de experiência, dependendo de como a recompensa é configurada).
  + É necessário adicionar lógica adicional nesta função para realmente conceder a recompensa ao jogador, como adicionar um item ao inventário, incrementar pontos de experiência, ou outras ações relacionadas.

**3. Fluxo e Interação**

1. **Missão Criada**: Quando o jogador recebe uma missão, ela é criada como uma instância da classe Mission. Ele verá o título e a descrição dessa missão e precisará interagir com o NPC correto, conforme o npc\_name, para poder completá-la.
2. **Verificação de Conclusão**: O jogador pode checar se completou a missão chamando o método check\_completion(). Isso permite ao jogador saber se a missão está pendente ou já foi concluída.
3. **Completando a Missão**: Uma vez que o jogador cumpra os requisitos da missão, o método complete\_mission() é chamado. Ele marca a missão como concluída, adiciona a missão à lista de missões completadas do jogador e chama give\_reward() para conceder a recompensa.
4. **Concedendo Recompensas**: A recompensa definida pela missão é dada ao jogador por meio do método give\_reward(), que pode ser expandido para adicionar lógica mais detalhada de recompensas, como a atribuição de itens ou a entrega de experiência.

**Possíveis Melhorias ou Expansões**

* **Missões Dependentes de Emoções de NPC**: Como você mencionou anteriormente, o NPC pode ter uma emoção específica necessária para completar a missão. Isso poderia ser integrado com a verificação de emoções no NPC, para que a missão só seja completada quando o NPC atingir uma determinada emoção.
* **Tipos de Recompensas**: A recompensa pode ser expandida para ser um objeto mais complexo, como uma classe Reward, que poderia incluir múltiplos tipos de recompensas (itens, moedas, experiência).
* **Interface de Missões**: Você poderia criar uma interface de missões no jogo, onde o jogador poderia acompanhar o progresso das missões, ver quais estão pendentes e quais foram completadas.

**Classe savedGame**:

**1. Variáveis e Referências**

* **@export var player\_position: Vector3**
  + **Tipo**: Vector3
  + **Função**: Armazena a posição do jogador no mundo 3D. Esse valor será usado para restaurar a localização exata do jogador quando o jogo for carregado.
* **@export var inventory: InventoryData**
  + **Tipo**: InventoryData (provavelmente uma classe que gerencia os itens do inventário do jogador)
  + **Função**: Armazena os dados do inventário do jogador, incluindo os itens que ele possui. Esse dado será usado para restaurar os itens do inventário quando o jogo for carregado.
* **@export var equip\_inventory: InventoryDataEquip**
  + **Tipo**: InventoryDataEquip (provavelmente uma classe que lida com itens equipados)
  + **Função**: Armazena os dados do inventário de equipamentos do jogador. Este dado é importante para restaurar os itens que o jogador está usando, como armas, armaduras, ou outros itens equipáveis.
* **@export var money: float**
  + **Tipo**: float
  + **Função**: Armazena a quantidade de dinheiro que o jogador tem. Esse valor pode ser usado para restaurar o saldo de dinheiro do jogador quando o jogo for carregado.
* **@export var missions\_complete := []**
  + **Tipo**: Array (provavelmente uma lista de missões completadas, armazenando o estado das missões)
  + **Função**: Armazena as missões que o jogador completou. Isso pode ser utilizado para restaurar o progresso do jogador em relação às missões quando o jogo for carregado. A variável é inicializada com uma lista vazia [], indicando que, por padrão, nenhuma missão foi completada.
* **@export var object\_name: String**
  + **Tipo**: String
  + **Função**: Armazena o nome associado ao objeto do jogo. Isso pode ser útil para identificar o arquivo salvo ou para fornecer um título identificável ao estado salvo.

**2. Fluxo e Interação**

* **Salvamento de Dados**: Quando o jogador opta por salvar o jogo, as variáveis da classe SavedGame armazenam o estado atual do jogo. Isso inclui a posição do jogador no mundo, os itens no inventário, os equipamentos, o dinheiro e as missões completadas. Esses dados são salvos em um arquivo de jogo, que pode ser carregado mais tarde.
* **Carregamento de Dados**: Quando o jogador carrega o jogo, os dados salvos são restaurados. A posição do jogador no mundo é definida de acordo com o valor em player\_position, o inventário e os itens equipados são restaurados, e o progresso nas missões e o dinheiro do jogador também são restaurados.

**3. Expansão Possível**

* **Tipos de Dados de Missões**: Atualmente, a variável missions\_complete é uma lista simples. No entanto, você poderia expandir isso para armazenar dados mais detalhados sobre cada missão, como a data de conclusão, condições especiais ou recompensas adicionais.
* **Validação de Dados**: Para garantir que os dados carregados sejam válidos (como um inventário com itens que o jogador pode usar), você poderia adicionar algum tipo de validação ao carregar o jogo. Isso garantiria que o jogo não falhe ao carregar dados corrompidos ou inválidos.
* **Suporte para Vários Arquivos de Salvamento**: Se o jogo oferecer a possibilidade de salvar múltiplos arquivos (por exemplo, para diferentes perfis de jogador ou slots de salvamento), você poderia adicionar suporte para gerenciar diferentes instâncias da classe SavedGame.

**Exemplo de Fluxo de Salvamento/Carregamento**

1. **Salvar o Jogo**: Quando o jogador escolhe salvar, você cria uma instância de SavedGame, preenche todas as variáveis com o estado atual do jogo (posição, inventário, dinheiro, missões), e então grava esse objeto em um arquivo.
2. **Carregar o Jogo**: Quando o jogador opta por carregar um jogo salvo, você lê os dados do arquivo, cria uma nova instância de SavedGame e preenche as variáveis da classe com os dados carregados. O estado do jogo é então restaurado, e o jogador retoma de onde parou.