Documentação do Script CharMove

Visão Geral

O script `CharMove` é anexado a um personagem em um jogo Unity e controla vários aspectos do movimento do personagem, ataques, defesa, saúde e interação com o ambiente do jogo. Ele é escrito em C# e utiliza várias bibliotecas e componentes específicos do Unity.

Declaração da Classe

csharp

```
public class CharMove : MonoBehaviour
```

A classe `CharMove` herda da classe `MonoBehaviour`, que é a classe base para todos os scripts do Unity que precisam interagir com GameObjects.

Variáveis de Membro

Variáveis Públicas

- `bool isDialoguing`: Indica se o personagem está atualmente envolvido em um diálogo. Por padrão, é definido como `false`.
- `bool canMove`: Indica se o personagem pode se mover. Por padrão, é definido como `true`.
- `GameObject interactableObject`: Uma referência ao GameObject com o qual o personagem pode interagir.
- `GameObject visualEffect`: Uma referência a um GameObject de efeito visual.
- `string SlipperyTag`: A tag usada para identificar objetos escorregadios no jogo.
- `int actualLife`: A saúde/vida atual do personagem.
- `int life`: A saúde/vida máxima do personagem, definida como 100 por padrão.
- `int quantityJump`: O número de pulos que o personagem pode executar.
- `float moveSpeed`: A velocidade de movimento do personagem, definida como 5f por padrão.
- `int coin`: O número de moedas coletadas pelo personagem.
- `bool AbilityActive`: Indica se a habilidade especial do personagem está ativa.
- `int SwordDamage`: O dano infligido pelo ataque de espada do personagem, definido como 10 por padrão.

- `int ArrowDamage`: O dano infligido pelo ataque de flecha do personagem, definido como 5 por padrão.
- `int ArrowMax`: O número máximo de flechas que o personagem pode carregar, definido como 10 por padrão.
- `float maxDistance`: A distância máxima para os ataques.
- `Image lifebar`: Referência à imagem UI representando a barra de vida do personagem.
- `Image redbar`: Referência à imagem UI representando a porção vermelha da barra de vida.
- `int lifeToRecover`: A quantidade de saúde a ser recuperada ao coletar um item de saúde, definida como 20 por padrão.
- `GameObject Ability`: Um GameObject representando a habilidade especial do personagem.
- `GameObject LightAttackChar`: Um GameObject representando o ataque leve do personagem.
- `GameObject HeavyAttackChar`: Um GameObject representando o ataque pesado do personagem.
- `GameObject ShieldChar`: Um GameObject representando o escudo do personagem.
- `TutorialController TC`: Referência ao script do controlador de tutorial.
- `PhaseManager PM`: Referência ao script do gerenciador de fases.
- `GameObject CameraUp`: Referência a um GameObject de câmera.
- `GameObject MyCam`: Referência ao GameObject da câmera principal.
- `Animator Anim`: Referência ao componente Animator do personagem.
- `Rigidbody Rb`: Referência ao componente Rigidbody do personagem.
- `BoxCollider Bc`: Referência ao componente BoxCollider do personagem.
- `AudioSource ASource`: Referência ao componente AudioSource do personagem.
- `AudioClip AC`: Um áudio clip a ser reproduzido quando o personagem pula.

Variáveis Privadas

- `int jumpsLeft`: O número de pulos restantes para o personagem.
- `Vector3 movement`: Um vetor para armazenar o movimento do personagem.
- `Vector3 scale`: A escala inicial do componente RectTransform da porção vermelha da barra de vida.
- `Vector3 newScale`: A escala desejada para a porção vermelha da barra de vida ao perder saúde.

Métodos

Start()

csharp

```
void Start()
```

O método `Start()` é chamado uma vez quando o script é inicializado. Ele é usado para inicializar as referências a componentes e configurar variáveis.

Update()

csharp

```
void Update()
```

O método `Update()` é chamado a cada quadro do jogo. Ele é usado para lidar com a entrada do jogador e chamar métodos correspondentes para mover o personagem, realizar ataques, defesa e outras ações.

FixedUpdate()

csharp

```
void FixedUpdate()
```

O método `FixedUpdate()` é chamado em intervalos fixos de tempo e é usado para lidar com a física do personagem. Ele é responsável por mover o personagem com base na entrada do jogador. Calcula a direção do movimento com base nos eixos de entrada, define a velocidade do personagem e reproduz as animações correspondentes.

Jump()

csharp

```
void Jump()
```

O método `Jump()` lida com o comportamento de pulo do personagem. Ele detecta a tecla de espaço pressionada e aplica uma força para cima ao componente Rigidbody do personagem. Ele também controla o número de pulos disponíveis.

LightAttack()

csharp

```
void LightAttack()
```

O método `LightAttack()` lida com o comportamento de ataque leve do personagem. Ele detecta o clique do botão esquerdo do mouse e aciona a animação e função de ataque correspondente.

HeavyAttack()

csharp

```
void HeavyAttack()
```

O método `HeavyAttack()` lida com o comportamento de ataque pesado do personagem. Ele detecta a combinação da tecla de controle esquerda pressionada e o clique do botão esquerdo do mouse e aciona a animação e função de ataque correspondente.

Attack()

csharp

```
private void Attack()
```

O método `Attack()` é chamado quando o personagem realiza um ataque. Ele calcula a posição e direção do ataque e verifica a colisão com objetos dentro do alcance `maxDistance`.

Defend()

csharp

```
void Defend()
```

O método `Defend()` lida com o comportamento de defesa do personagem. Ele detecta o clique do botão direito do mouse e define o parâmetro de animação correspondente.

OnTriggerEnter(Collider other)

csharp

```
public void OnTriggerEnter(Collider other)
```

O método `OnTriggerEnter(Collider other)` é chamado quando o personagem entra em um colisor de gatilho. Ele lida com interações com vários objetos do jogo, como coletar moedas, acionar diálogos e mudar de fase.

OnTriggerExit(Collider other)

csharp

```
private void OnTriggerExit(Collider other)
```

O método `OnTriggerExit(Collider other)` é chamado quando o personagem sai de um colisor de gatilho. Ele é usado para redefinir a variável `canMove` quando o personagem não está mais em contato com um objeto interativo.

TakeDamage(int damage)

csharp

```
public void TakeDamage(int damage)
```

O método `TakeDamage(int damage)` é chamado quando o personagem recebe dano. Ele diminui a saúde do personagem pelo valor do dano e atualiza a barra de vida correspondente na interface do usuário. Se a saúde do personagem chegar a zero, ele chama o método `Die()`.

Die()

csharp

```
private void Die()
```

O método `Die()` é chamado quando a saúde do personagem chega a zero. Ele executa a lógica de morte do personagem, como a reprodução de uma animação de morte, a desativação do controle de movimento e a exibição de uma tela de game over.

Melhorias Sugeridas

- Utilizar propriedades (getters e setters) para acessar e modificar as variáveis de membro, em vez de torná-las públicas.
- Dividir o código em várias classes e scripts menores para melhorar a modularidade e a legibilidade.

- Utilizar o padrão de design Observer para lidar com eventos e interações entre objetos do jogo.
- Refatorar a lógica de detecção de colisão e interação para torná-la mais eficiente e flexível.
- Adicionar comentários e documentação detalhada em cada método e classe para facilitar a compreensão e manutenção do código.
- Utilizar constantes ou enums para evitar o uso de valores literais diretamente no código.
- Melhorar a organização e nomenclatura das variáveis para tornar o código mais legível e de fácil manutenção.
- Utilizar o recurso de pooling de objetos para melhorar o desempenho ao lidar com a criação e destruição de GameObjects repetidamente.
- Utilizar animações de transição suaves para tornar as animações do personagem mais realistas e fluidas.
- Separar a lógica do personagem (movimento, ataques) da lógica do ambiente (coleta de moedas, interações com objetos) em scripts distintos.
- Utilizar eventos e delegados para permitir a comunicação entre diferentes scripts sem acoplamento direto.
- Implementar um sistema de save/load para manter o estado do personagem e permitir que o jogador retome o jogo a partir do ponto em que parou.
- Adicionar tratamento de erros e exceções para evitar que o jogo trave ou tenha comportamento inesperado em situações de erro.
- Utilizar um sistema de gerenciamento de áudio para controlar a reprodução de sons e músicas no jogo.
- Realizar testes unitários e de integração para garantir a qualidade e funcionamento correto do código.

Para melhorar a legibilidade e modularidade do código, é possível criar as seguintes subclasses:

1. `PlayerMovement` (Subclasse de `MonoBehaviour`):
    - Responsável pelo movimento do personagem.
    - Lidaria com a detecção e processamento dos inputs do jogador para mover o personagem.
    - Poderia conter métodos como `Move()`, `Rotate()`, `HandleInput()`.
2. `PlayerCombat` (Subclasse de `MonoBehaviour`):
    - Responsável pelos ataques e defesa do personagem.
    - Lidaria com a detecção de inputs do jogador para realizar ataques e defesa.
    - Poderia conter métodos como `LightAttack()`, `HeavyAttack()`, `Defend()`.
3. `PlayerHealth` (Subclasse de `MonoBehaviour`):
    - Responsável pela saúde do personagem.
    - Lidaria com a lógica de controle da saúde do personagem, como receber dano e verificar se o personagem está vivo.
    - Poderia conter métodos como `TakeDamage()`, `Die()`.
4. `InteractableObject` (Subclasse de `MonoBehaviour`):
    - Responsável pelos objetos interativos do jogo.

    - Lidaria com a interação do personagem com objetos, como coletar moedas, acionar diálogos, etc.
    - Poderia conter métodos como `Interact()`.
5. `Enemy` (Subclasse de `MonoBehaviour`):
    - Responsável pelos inimigos do jogo.
    - Lidaria com o comportamento e a lógica dos inimigos, como movimento, ataques, detecção de colisões, etc.
    - Poderia conter métodos como `Move()`, `Attack()`, `TakeDamage()`, `Die()`.

Essas subclasses ajudariam a organizar o código de forma mais clara, separando as responsabilidades e melhorando a legibilidade e a manutenção. Além disso, permitiriam reutilizar funcionalidades comuns entre personagem e inimigos, evitando duplicação de código.

TutorialController

O script TutorialController é responsável pelo controle de um tutorial específico do jogo. Ele lida com a exibição de um botão de tutorial e a ativação/desativação de um objeto de tutorial quando o jogador pressiona uma determinada tecla.

Variáveis de Membro:

- **isTutoE** (bool): Indica se o tutorial do botão E está ativo.
- **TutoE** (GameObject): Referência ao objeto de tutorial do botão E.

Métodos:

- **Start()**: Método chamado antes do primeiro quadro. Responsável por inicializar o tutorial.
- **Update()**: Método chamado a cada quadro. Verifica se o jogador pressionou a tecla E e o tutorial do botão E não está ativo.
- **ShowButton()**: Alterna a exibição do tutorial do botão E.

PhaseManager

O script PhaseManager é responsável pelo gerenciamento das fases do jogo. Ele lida com o carregamento de fases, avanço de fase, acesso à fase salva, configurações, créditos e abertura de formulários externos.

Variáveis de Membro:

- **ActualPhase** (int): Indica a fase atual do jogo.

Métodos:

- **Start()**: Método chamado antes do primeiro quadro. Verifica se a fase atual está dentro de um intervalo específico (1 a 5) e, se estiver, armazena o valor da fase atual no PlayerPrefs.
- **Fase(int numberPhase)**: Carrega a fase especificada pelo número.
- **PassPhase()**: Avança para a próxima fase.
- **GoToSavedPhase()**: Carrega a fase salva no PlayerPrefs.
- **Settings()**: Método vazio. Pode ser implementado para lidar com as configurações do jogo.
- **Credits()**: Carrega a cena de créditos.
- **OpenForms()**: Abre um formulário externo usando a função Application.OpenURL().

CameraShake

O script CameraShake é responsável por criar um efeito de trepidação na posição da câmera.

Variáveis de Membro:

- **shakeDuration** (float): A duração da trepidação em segundos.
- **shakeMagnitude** (float): A intensidade da trepidação.

Métodos:

- **Shake()**: Inicia a trepidação da câmera.
- **ShakeCoroutine()**: Corrotina que realiza a trepidação da câmera. Calcula novas posições aleatórias dentro de um intervalo especificado e atualiza a posição da câmera. A corrotina é executada durante um intervalo de tempo determinado por shakeDuration.

Credits

O script Credits é responsável por exibir créditos rolando na tela e retornar para a primeira cena após a conclusão dos créditos.

Variáveis de Membro:

- **textComponent** (Text): Referência ao componente de texto que contém os créditos.
- **scrollSpeed** (float): A velocidade de rolagem dos créditos.
- **firstSceneName** (string): O nome da primeira cena a ser carregada após a conclusão dos créditos.

Métodos:

- **Start()**: É chamado no início do jogo. Obtém a referência ao componente RectTransform do texto e define a posição inicial de rolagem dos créditos.

- **Update()**: É chamado a cada quadro. Chama o método ScrollText() para rolar os créditos e verifica se a posição de rolagem atingiu o limite máximo. Se atingir, chama o método ReturnToFirstScene() para retornar à primeira cena.
- **ScrollText()**: Realiza a rolagem dos créditos, atualizando a posição do componente RectTransform do texto. A posição de rolagem é incrementada com base na velocidade de rolagem e no tempo decorrido desde o último quadro. Quando a posição de rolagem atinge o limite máximo (altura do texto), ela é reiniciada para o início.
- **ReturnToFirstScene()**: Carrega a primeira cena especificada no nome firstSceneName, reiniciando assim o jogo.

UISettings

O script UISettings é responsável por controlar as configurações do usuário na interface do usuário de configurações. Ele utiliza barras de rolagem para ajustar a sensibilidade do mouse e o volume do som.

Variáveis de Membro:

- **mouseSensitivityScrollbar** (Scrollbar): Referência à barra de rolagem que controla a sensibilidade do mouse.
- **soundVolumeScrollbar** (Scrollbar): Referência à barra de rolagem que controla o volume do som.
- **configManager** (ConfigManager): Referência ao gerenciador de configurações.

Métodos:

- **Start()**: É chamado no início do jogo. Obtém a referência ao ConfigManager por meio do método GameObject.FindObjectOfType<ConfigManager>(). Em seguida, adiciona ouvintes aos eventos onValueChanged das barras de rolagem para chamar os métodos de atualização de sensibilidade do mouse e volume do som no ConfigManager.

ConfigManager

O script ConfigManager é responsável por gerenciar as configurações do jogo, como sensibilidade do mouse e volume do som.

Variáveis de Membro:

- **mouseSensibility** (float): A sensibilidade do mouse.
- **soundVolume** (float): O volume do som.
- **CC** (CameraController): Referência ao controlador da câmera.

CameraController

O script CameraController é responsável por controlar a rotação da câmera do jogador com base nos movimentos do mouse.

Variáveis de Membro:

- **sensitivy** (float): A sensibilidade do mouse.
- **target** (Transform): O objeto alvo que a câmera deve seguir.
- **xRotation** (float): A rotação em torno do eixo x (vertical) da câmera.
- **yRotation** (float): A rotação em torno do eixo y (horizontal) da câmera.

Métodos:

- **Start()**: É chamado no início do jogo. Pode ser usado para configurar o estado inicial da câmera, como travar o cursor.
- **Update()**: É chamado a cada quadro. Calcula os movimentos do mouse (Input.GetAxis) e atualiza a rotação da câmera com base nesses movimentos. A rotação horizontal é aplicada ao objeto alvo (target) se estiver definido. A rotação vertical (xRotation) é aplicada à própria câmera (transform.localRotation).

Métodos:

- **Start()**: É chamado no início do jogo. Define a sensibilidade do mouse com base no valor da variável "sensitivy" do controlador da câmera (CC).
- **UpdateMouseSensitivity(float newValue)**: Atualiza a sensibilidade do mouse com o novo valor fornecido.
- **UpdateSoundVolume(float newValue)**: Atualiza o volume do som com o novo valor fornecido. Geralmente, essa função aplicaria o novo volume ao áudio do jogo usando algo como "Audio.Listener.volume = soundVolume".

GameController

O script GameController é responsável por controlar o fluxo de jogo e interações do jogador, como pausar o jogo, acessar o menu de configurações e realizar ações como reiniciar o jogo e sair do jogo.

Variáveis de Membro:

- **pauseMenuUI** (GameObject): Referência ao objeto que representa o menu de pausa no jogo.

- **pauseMenuCanvas** (GameObject): Referência ao objeto que contém o menu de pausa.
- **isPaused** (bool): Indica se o jogo está pausado ou não.
- **Settings** (GameObject): Referência ao objeto que representa o menu de configurações.
- **resumeButton** (GameObject): Referência ao botão "Resume" no menu de pausa.

Métodos:

- **Update()**: É chamado a cada quadro. Verifica se a tecla de escape (KeyCode.Escape) foi pressionada. Se o jogo estiver pausado, chama o método Resume() para retomar o jogo. Se o menu de configurações estiver ativo, fecha o menu. Caso contrário, chama o método Pause() para pausar o jogo.
- **Resume()**: Desativa o menu de pausa, restaura o tempo (Time.timeScale = 1f) e define isPaused como false, indicando que o jogo está sendo retomado.
- **Pause()**: Ativa o menu de pausa, congela o tempo (Time.timeScale = 0f) e define isPaused como true, indicando que o jogo está pausado.
- **ResetGame()**: Reinicia o jogo carregando a cena atual usando SceneManager.LoadScene. Chama o método Resume() para retomar o jogo após o reinício.
- **QuitGame()**: Sai do jogo, encerrando a aplicação.
- **GoToFirstScene()**: Carrega a primeira cena do jogo usando SceneManager.LoadScene. Chama o método Resume() para retomar o jogo após a transição.
- **GoToSettings()**: Ativa o menu de configurações e desativa o menu de pausa.
- **ResumeGameFromSettings()**: Desativa o menu de configurações e chama o método Resume() para retomar o jogo.
- **Clicou()**: Um método de exemplo que imprime uma mensagem no console quando um botão é clicado.

CoinToString

O script CoinToString é responsável por atualizar o texto de um componente Text com o valor atual da moeda obtida pelo jogador.

Variáveis de Membro:

- **CM** (CharMove): Referência ao script CharMove que contém a função GetCoin().
- **MyText** (Text): Referência ao componente Text que exibirá o valor da moeda.

Métodos:

- **Update()**: É chamado a cada quadro. Atualiza o texto de MyText com o valor retornado pela função GetCoin() do script CharMove, convertendo-o para uma string.

LifeToString

O script LifeToString é responsável por atualizar o texto de um componente Text com o valor atual da vida do jogador.

Variáveis de Membro:

- **CM** (CharMove): Referência ao script CharMove que contém a função GetHealth().
- **MyText** (Text): Referência ao componente Text que exibirá o valor da vida.

Métodos:

- **Update()**: É chamado a cada quadro. Atualiza o texto de MyText com o valor retornado pela função GetHealth() do script CharMove, convertendo-o para uma string.

MovableRock

O script MovableRock é responsável por permitir que o jogador empurre pedras móveis no ambiente do jogo.

Variáveis de Membro:

- **moveSpeed** (float): A velocidade de movimento da pedra.
- **maxDistance** (float): A distância máxima que a pedra pode percorrer antes de parar.
- **characterAnimator** (Animator): Referência ao componente Animator do personagem que está empurrando a pedra.
- **obstacleLayer** (LayerMask): Camadas consideradas obstáculos para a detecção de colisão.

Métodos:

- **Start()**: É chamado no início do jogo. Obtém referências aos componentes Rigidbody e ao GameObject do jogador.
- **Update()**: É chamado a cada quadro. Move a pedra na direção definida pelo jogador, desde que não haja obstáculos bloqueando o caminho.
- **OnTriggerEnter(Collider other)**: É chamado quando o jogador entra na área de colisão da pedra. Ativa o movimento da pedra e define a referência para a pedra atualmente sendo empurrada.
- **OnTriggerExit(Collider other)**: É chamado quando o jogador sai da área de colisão da pedra. Interrompe o movimento da pedra.
- **StopMovingRock()**: Interrompe o movimento da pedra, define as propriedades corretas e limpa a referência para a pedra atualmente sendo empurrada.

Slippery

O script Slippery permite que o personagem deslize em superfícies escorregadias.

Variáveis de Membro:

- **slipperyTag** (string): A tag atribuída aos objetos escorregadios no jogo.

Métodos:

- **Start()**: É chamado no início do jogo. Obtém uma referência ao componente Rigidbody do personagem.
- **FixedUpdate()**: É chamado a cada quadro fixo. Permite que o personagem se mova horizontalmente e verticalmente quando está em uma superfície escorregadia.
- **OnCollisionEnter(Collision collision)**: É chamado quando o personagem colide com outro objeto. Verifica se o personagem colidiu com uma superfície escorregadia ou uma pedra e reage adequadamente.

MovablePlatform

O script MovablePlatform permite que uma plataforma se mova de forma contínua para frente e para trás ao longo de um eixo selecionado.

Variáveis de Membro:

- **moveSpeed** (float): A velocidade de movimento da plataforma.
- **distance** (float): A distância total que a plataforma percorrerá antes de inverter a direção.
- **movementAxis** (Axis): O eixo ao longo do qual a plataforma se moverá (X, Y ou Z).

Métodos:

- **Start()**: É chamado no início do jogo. Armazena a posição inicial da plataforma.
- **Update()**: É chamado a cada quadro. Move a plataforma para frente e para trás ao longo do eixo selecionado. Inverte a direção do movimento quando a distância percorrida atinge a distância total.

Enumeração:

- **Axis**: Representa os eixos possíveis ao longo dos quais a plataforma pode se mover (X, Y ou Z).

Editor Personalizado:

O script também possui um editor personalizado que permite editar as propriedades do script no Editor do Unity de forma mais conveniente.

Variáveis de Membro:

- **cm** (CharMove): Referência ao script "CharMove" do personagem do jogador.
- **target** (Transform): Transform do personagem do jogador que está sendo perseguido.
- **moveSpeed** (float): Velocidade de movimento do Beholder.
- **rb** (Rigidbody): Componente Rigidbody do Beholder.
- **animator** (Animator): Componente Animator do Beholder.
- **initialPosition** (Vector3): Posição inicial do Beholder.
- **isChasing** (bool): Indica se o Beholder está perseguindo o personagem do jogador.
- **ps** (ParticleSystem): Referência a um ParticleSystem para reproduzir partículas quando o Beholder tomar dano.
- **enemyName** (string): Nome do inimigo Beholder.
- **attackDamage** (int): Quantidade de dano causado pelo ataque do Beholder.
- **health** (int): Quantidade de pontos de vida do Beholder.
- **speed** (int): Velocidade do Beholder.
- **attackRange** (float): Alcance de ataque do Beholder.
- **dodgeChance** (float): Chance de esquiva do Beholder.
- **followRadius** (float): Raio de distância dentro do qual o Beholder começa a seguir o personagem do jogador.
- **hitForce** (float): Força de empurrão aplicada ao Beholder quando ele toma dano.

Métodos:

- **Start()**: Chamado no início do jogo. Obtém as referências aos componentes Rigidbody e Animator e armazena a posição inicial do Beholder.
- **Awake()**: Chamado quando o Beholder é ativado. Define os valores iniciais para as variáveis attackDamage e health.
- **Update()**: Chamado a cada quadro. Controla o comportamento do Beholder, incluindo a perseguição do personagem do jogador, o ataque quando estiver dentro do alcance e as animações correspondentes.
- **OnTriggerEnter(Collider other)**: Chamado quando o Beholder colide com um objeto. Verifica se colidiu com o personagem do jogador para iniciar a perseguição, aplica dano ao personagem quando o Beholder ataca, e reduz a saúde do Beholder quando atingido por um ataque do personagem.
- **Die()**: Chamado quando a saúde do Beholder chega a zero. Reproduz a animação de morte e destrói o objeto Beholder após um tempo.

`DialogueController`

Controla a exibição de diálogos em um painel de diálogo. Ele exibe o texto de fala do personagem, a imagem do perfil e o nome do ator. O texto é exibido gradualmente, como se estivesse sendo digitado, e o jogador pode avançar para o próximo diálogo.

Variáveis de Membro:

- **dialogueObj** (GameObject): Referência ao objeto de diálogo.
- **profile** (Image): Componente Image que exibe a imagem do perfil do personagem.

- **speechText** (TextMeshProUGUI): Componente TextMeshProUGUI que exibe o texto de fala.
- **actorNameText** (TextMeshProUGUI): Componente TextMeshProUGUI que exibe o nome do ator.
- **typingSpeed** (float): Velocidade de digitação do texto.
- **sentences** (string[]): Array de strings contendo as sentenças de diálogo.
- **index** (int): Índice atual da sentença de diálogo.
- **isTyping** (bool): Indica se o texto está sendo digitado.
- **isTextComplete** (bool): Indica se o texto foi completamente exibido.
- **typingCoroutine** (Coroutine): Referência à coroutine de digitação.

Métodos:

- **Speech(Sprite p, string[] txt, string actorName)**: Inicia a exibição de um novo diálogo. Recebe o sprite do perfil do personagem, um array de strings contendo as sentenças do diálogo e o nome do ator. Ativa o painel de diálogo, define o sprite do perfil, as sentenças e o nome do ator. Chama a coroutine `TypeSentences()` para exibir o texto gradualmente.
- **TypeSentences()**: Coroutine que exibe o texto gradualmente, como se estivesse sendo digitado. Cada letra é adicionada ao texto com um pequeno atraso, determinado pela variável `typingSpeed`.
- **NextSentences()**: Avança para a próxima sentença de diálogo. Se o texto estiver sendo digitado, a digitação é interrompida e o texto completo é exibido. Se o texto estiver completo, verifica se há mais sentenças de diálogo. Se houver, incrementa o índice e chama a coroutine `TypeSentences()` novamente. Caso contrário, limpa o texto, redefine o índice e desativa o painel de diálogo.

`Dialogue`

É responsável por ativar o diálogo quando o jogador entra na área de interação. Ele detecta a proximidade do jogador usando um colisor em forma de esfera e, quando o jogador pressiona a tecla "E" dentro da área de interação, ele chama o método `Speech` do `DialogueController` para exibir o diálogo.

Variáveis de Membro:

- **profile** (Sprite): Sprite do perfil do personagem associado ao diálogo.
- **speechTxt** (string[]): Array de strings contendo as sentenças do diálogo.
- **actorName** (string): Nome do ator associado ao diálogo.
- **radius** (int): Raio da área de interação.
- **playerLayer** (LayerMask): Layer usada para identificar o jogador.
- **controller** (DialogueController): Referência ao componente `DialogueController`.
- **proximityDistance** (float): Distância de proximidade para ativar o diálogo.
- **onRadius** (bool): Indica se o jogador está dentro da área de interação.

Métodos:

- **Start()**: Chamado no início do jogo. Obtém uma referência ao componente `DialogueController`.
- **Update()**: Chamado a cada quadro. Verifica se o jogador pressionou a tecla "E" e está dentro da área de interação. Se as condições forem atendidas, chama o método `Speech` do `DialogueController` para exibir o diálogo.
- **FixedUpdate()**: Chamado em intervalos fixos. Verifica se o jogador está dentro da área de interação usando uma distância de proximidade. Define o valor de `onRadius` com base no resultado da verificação.
- **Interact3D(float distance)**: Verifica se o jogador está dentro da área de interação usando uma distância de proximidade em um ambiente 3D. Obtém uma referência ao objeto do jogador usando a tag "Character" e calcula a distância entre o objeto atual e o jogador. Define o valor de `onRadius` com base na distância calculada.

`AbilitiesToBuy`

É responsável por gerenciar as habilidades disponíveis para compra pelo jogador. Cada habilidade tem um custo em moedas e, quando comprada, afeta os atributos do personagem controlado pelo jogador.

Variáveis de Membro:

- **character** (CharMove): Referência ao componente `CharMove` do personagem controlado pelo jogador.
- **YDNHM** (GameObject): Referência ao objeto que exibe a mensagem "You do not have enough money" (Você não tem dinheiro suficiente) quando o jogador não tem moedas suficientes para comprar uma habilidade.
- **YAB** (GameObject): Referência ao objeto que exibe a mensagem "You have already bought this ability" (Você já comprou essa habilidade) quando o jogador tenta comprar uma habilidade que já possui.
- **Berserk** (bool): Indica se a habilidade Berserk foi comprada.
- **DeathKnight** (bool): Indica se a habilidade DeathKnight foi comprada.
- **Hunter** (bool): Indica se a habilidade Hunter foi comprada.
- **Ranger** (bool): Indica se a habilidade Ranger foi comprada.
- **Space** (bool): Indica se a habilidade Space foi comprada.
- **Druid** (bool): Indica se a habilidade Druid foi comprada.
- **Rage** (bool): Indica se a habilidade Rage foi comprada.

Métodos:

- **aBerserk()**: Chamado quando o jogador tenta comprar a habilidade Berserk. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **aDeathKnight()**: Chamado quando o jogador tenta comprar a habilidade DeathKnight. Verifica se o jogador tem moedas suficientes e se a habilidade ainda

não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.

- **aHunter()**: Chamado quando o jogador tenta comprar a habilidade Hunter. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **aRanger()**: Chamado quando o jogador tenta comprar a habilidade Ranger. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **aSpace()**: Chamado quando o jogador tenta comprar a habilidade Space. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **aDruid()**: Chamado quando o jogador tenta comprar a habilidade Druid. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **aRage()**: Chamado quando o jogador tenta comprar a habilidade Rage. Verifica se o jogador tem moedas suficientes e se a habilidade ainda não foi comprada. Se as condições forem atendidas, atualiza os atributos do personagem e subtrai o custo da habilidade das moedas do jogador.
- **DoNotHaveMoney()**: Coroutine que exibe a mensagem "You do not have enough money" (Você não tem dinheiro suficiente) por um curto período de tempo.
- **AlreadyBuy()**: Coroutine que exibe a mensagem "You have already bought this ability" (Você já comprou essa habilidade) por um curto período de tempo.
- **DeactivateThis()**: Desativa o objeto `YDNHM` que exibe a mensagem "You do not have enough money".
- **DeactivateThat()**: Desativa o objeto `YAB` que exibe a mensagem "You have already bought this ability".

Essas são as funcionalidades do script `AbilitiesToBuy`. Ele gerencia a compra de habilidades pelo jogador e atualiza os atributos do personagem de acordo com as habilidades compradas.