

Desenvolvendo "Foguinho e Aguinha" na Unity 3D

Animações, Multi-Animações, Triggers.

O jogo Foguinho e Aguinha é um jogo de plataforma cooperativo onde dois personagens, um de fogo e outro de água, precisam colaborar para superar desafios. Este tutorial guiará você pelo processo de desenvolvimento na Unity 3D (versão 6000), usando C# e a biblioteca padrão da Unity com suporte para até 4 jogadores.

Serão abordadas desde a configuração inicial do projeto até a programação de movimentação, colisões, sistema de portais, botões de pressão, transição de fases e muito mais, com foco total em acessibilidade, clareza e estrutura profissional.

Configuração do Projeto

Criar o projeto

1. Abra o Unity Hub e crie um novo projeto do tipo 2D.
2. Nomeie como *Elements* e escolha a pasta de destino.
3. Clique em Create.

Ajustes Iniciais

1. Na janela Game, defina a resolução para 1920x1080.
2. Vá em *File > Build Settings* e adicione a cena atual com o botão *Add Open Scenes*.
3. Em *Edit > Project Settings > Time*, defina *Fixed Timestep* para 0.02.

Importando Assets

1. Crie pastas: Sprites, Scripts, Scenes, Animations, InputActions, Prefabs.
2. Importe suas imagens de personagens, blocos, portais e portas para a pasta Sprites.
3. Usando o Sprite Editor, verifique se as imagens foram cortadas corretamente.
4. Modifique as configurações:
 - a. Filter Mode: Point
 - b. Compression: High Quality

Configuração do Input System

O novo Input System da Unity é a melhor escolha se você vai ter dois (ou mais) jogadores usando controles diferentes ao mesmo tempo. Se você está usando um controle de Xbox para PC na Unity, os botões e eixos do controle são acessados via `Input.GetAxis` ou `Input.GetButton`, não via `KeyCode`.

🎯 Por que usar o novo Input System?

- 🎮 Suporte real a múltiplos controles (Xbox, PS, teclado, etc).
- ✨ Cada jogador pode ter seu próprio dispositivo e esquema de input.
- ✨ Você pode usar `PlayerInput` + `Input Actions` para instanciar jogadores com controle atribuído automaticamente.
- 📞 Detecta dinamicamente quem está controlando quem.
- ✅ Ideal para local multiplayer, coop, jogos de luta, etc.

✅ Recomendado pra você se:

- Está fazendo um jogo multiplayer local com dois personagens.
- Quer que cada um use seu controle sem confusão.
- Quer um sistema limpo e extensível no futuro.

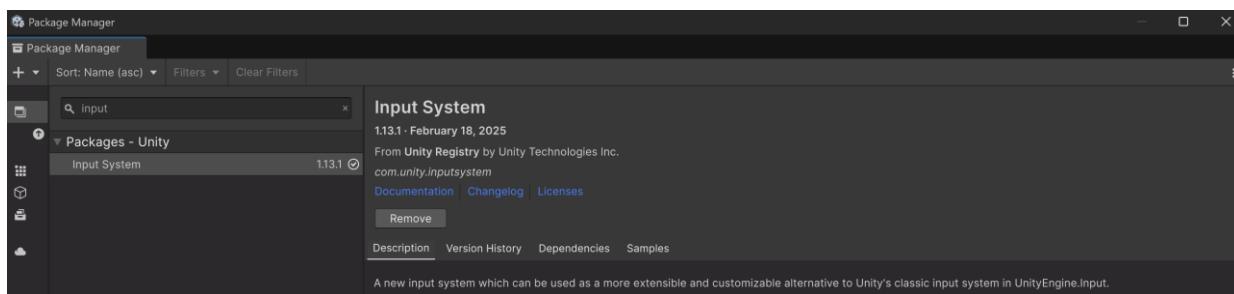
✨ Estrutura geral

- `PlayerInput` para detectar o controle e associar ao personagem.
- `Input Actions` com ações `Move (Vector2)` e `Jump (Button)`.
- Cada personagem vai usar o mesmo script, mas com `PlayerInput` configurado individualmente.
- `PlayerInputManager` vai instanciar automaticamente os jogadores (até 4).

Instale o Input System

Vamos instalar o pacote na Unity3D.

1. Vá no menu `<Window>` → `<<Package Manager>>`.
2. Selecione *Unity Registry*, procure por *Input System*.



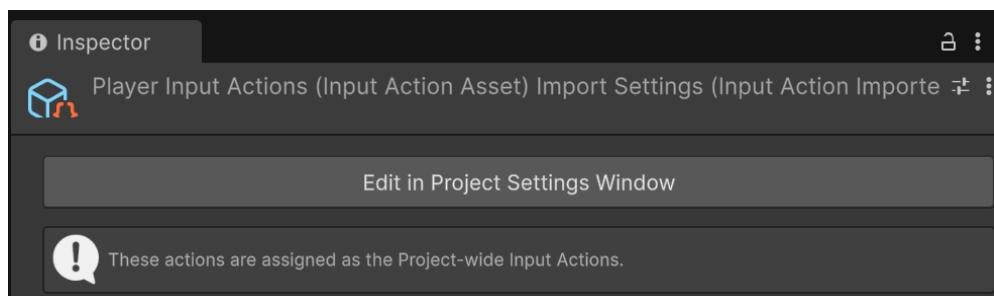
3. Clique em `Install`.

4. Após instalar, aceite a reinicialização da Unity.

Crie o Input Actions

Vamos agora configurar o controle.

1. Crie uma pasta *Assets/InputActions*.
2. Clique com o botão direito na pasta e vá em *Create → Input Actions*
3. Nomeie como: *PlayerInputActions*
4. Abra o editor clicando em *Edit in Project Setting Windows*



5. Crie um *Action Map* chamado *Gameplay*. Caso já existir, passe para o próximo passo.
6. Adicione as seguintes ações:

- Action Map: *Gameplay*
 - Action: Move
 - Type: Value
 - Control Type: Vector2
 - Bindings:
 - Gamepad → LeftStick
 - Keyboard → WASD (usando Composite 2D Vector)
 - Keyboard → Setas direcionais (LeftArrow, RightArrow, UpArrow, DownArrow)
 - Action: Jump
 - Type: Button
 - Bindings:
 - Gamepad → ButtonSouth (botão A do Xbox ou X do PS)
 - Keyboard → Space
 - Action: Next
 - Type: Button
 - Bindings:
 - Gamepad → Start
 - Keyboard → Enter
 - Action: Pause
 - Type: Button
 - Bindings:
 - Gamepad → Start

➤ Keyboard → Esc

7. Clique em *Save Asset*.

💡 Certifique-se que os nomes das ações (Move, Jump) estejam exatamente como os métodos em seu script: OnMove() e OnJump().

Configuração dos Sprites

Para facilitar a identificação dos sprites, é recomendado que você abra os sprites para renomear para um nome que o identifique.

1. Selecione os sprites que serão usados para a animação Idle, muito usado quando o personagem está parado sem fazer nada.

- a. Nome: UI_Player_WaterGirl_Idle_Body_01



- b. Nome: UI_Player_WaterGirl_Idle_Head_01 a UI_Player_WaterGirl_Idle_Head_08

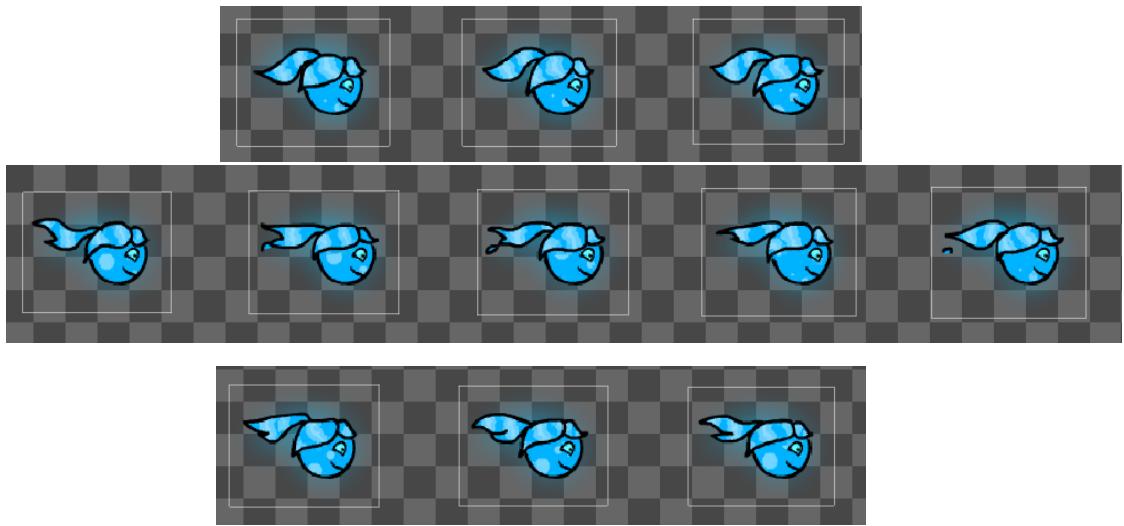


2. Selecione os sprites que demonstra a movimentação de andando.

- a. Nome: UI_Player_WaterGirl_Walking_Body_01 a UI_Player_WaterGirl_Walking_Body_08

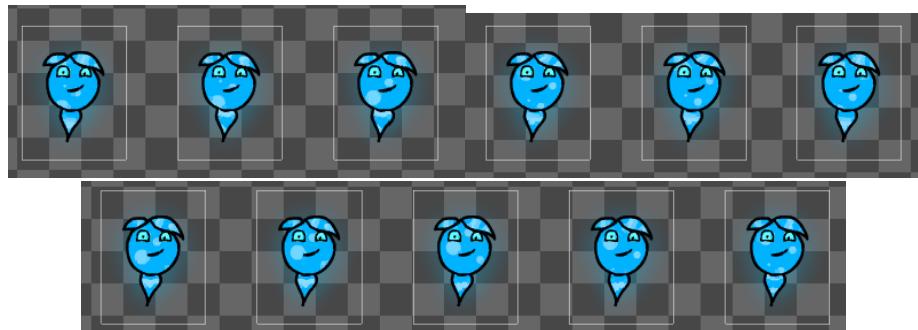


- b. Nome: UI_Player_WaterGirl_Walking_Head_01 a UI_Player_WaterGirl_Walking_Head_11



3. Selecione os sprites que demonstra a movimentação de morrendo.

- a. Nome: UI_Player_WaterGirl_Dying_Body_01 a UI_Player_WaterGirl_Dying_Body_11



4. Selecione os sprites que demonstra a movimentação de entrando na porta.

- a. Nome: UI_Player_WaterGirl_EnterDoor_FullBody_01 a
UI_Player_WaterGirl_EnterDoor_FullBody_13



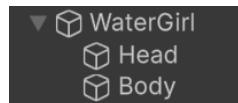
5. Selecione os sprites que demonstra as outras movimentações que você ache relevante e faça.

Criando os Personagens

Vamos criar inicialmente o personagem WaterGirl e depois o FireMan. Caso queira criar outros, o procedimento será o mesmo.

Montagem Estrutural

1. No *Hierarchy*, crie um objeto vazio e renomeie chamado WaterGirl.
2. Dentro dele, crie dois objetos filhos: Head e Body.

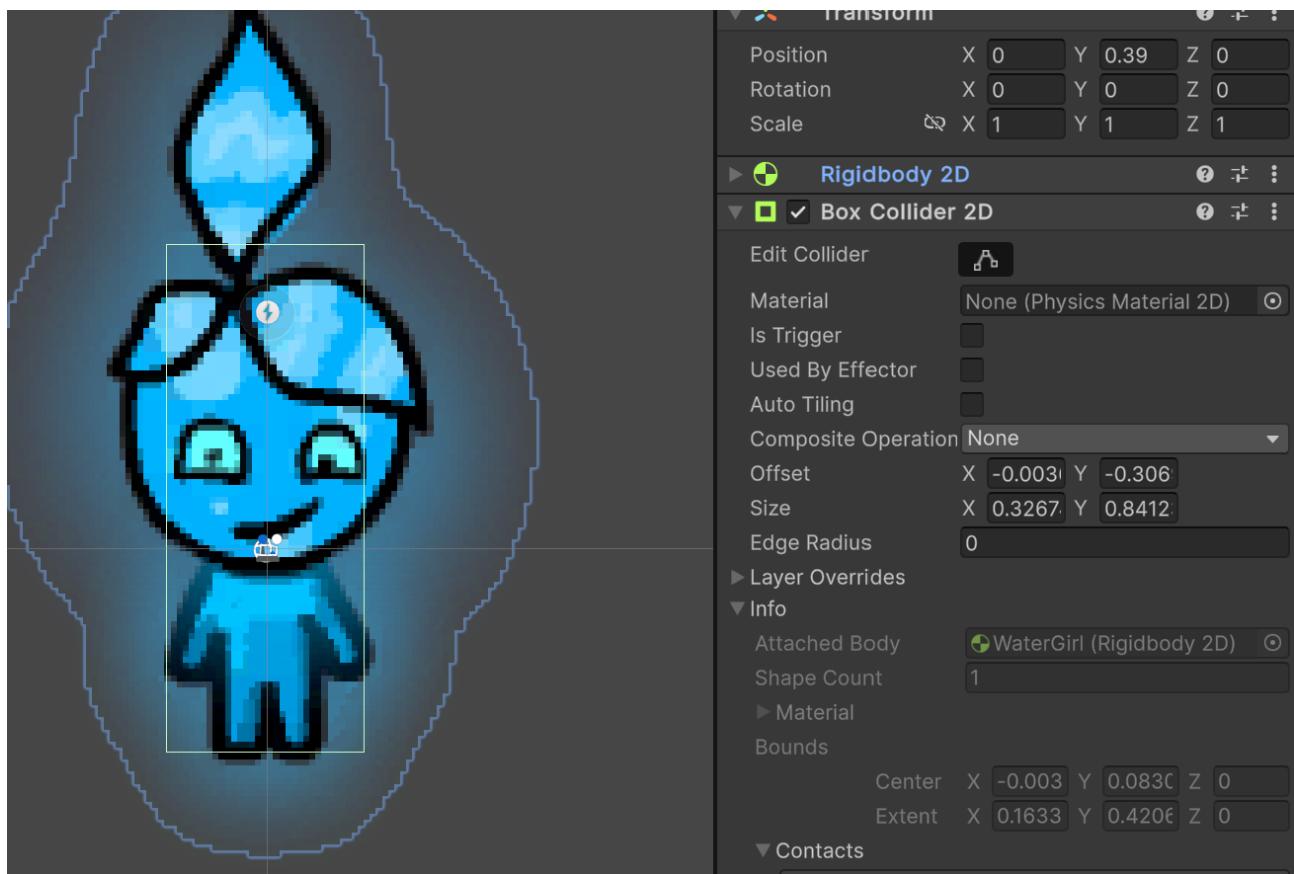


3. Adicione um Sprite Renderer no Head e Body e atribua a imagem correspondente na qual o personagem fique de frente.



4. Coloque Order in Layer = 1 para ambas as partes do corpo
5. Adicione um Rigidbody2D (modo "Dynamic") no *gameobject* WaterGirl.
6. Adicione um BoxCollider2D e modifique a colisão para cobrir toda a área do personagem.

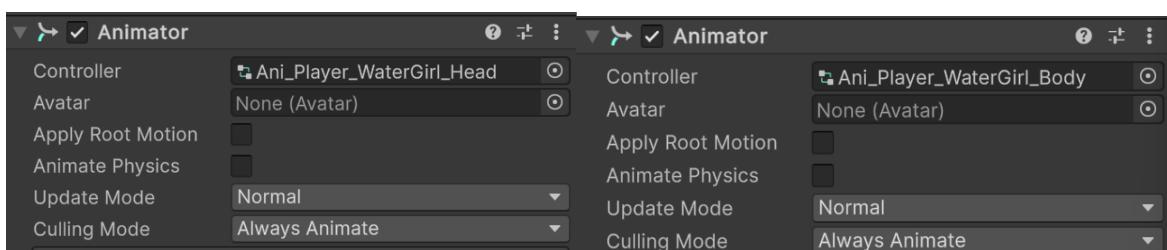
A melhor forma de se fazer é criar um collider na cabeça e o outro no corpo, mas isso será evitado para evitar maior complexidade na programação.



Configurando o Animator

Vamos criar agora as configurações iniciais para trabalhar com várias animações. Como a cabeça e o corpo estão separados, é essencial criar um animator para cada e as animações devem ser dessas partes.

1. Crie um Animator Controller para a Head e Body da WaterGirl.
2. Renomeei colocando os nomes com a seguinte regra:
 - a. AniC_[Player]_[NomePersonagem]_[ParteDoCorpo]
3. Adicione o componente Animator em cada um deles e atribua no campo Controller o Animator clicando no ícone de círculo.



4. Crie os seguintes estados nos parâmetros do Animator:
 - Walking (coloque bool isWalking)

- Jumping (coloque bool isJumping)
- Dying (coloque trigger isDead)
- EnterDoor (coloque bool isEnterDoor)

5. Crie todas as animações colocando os nomes com a seguinte regra:

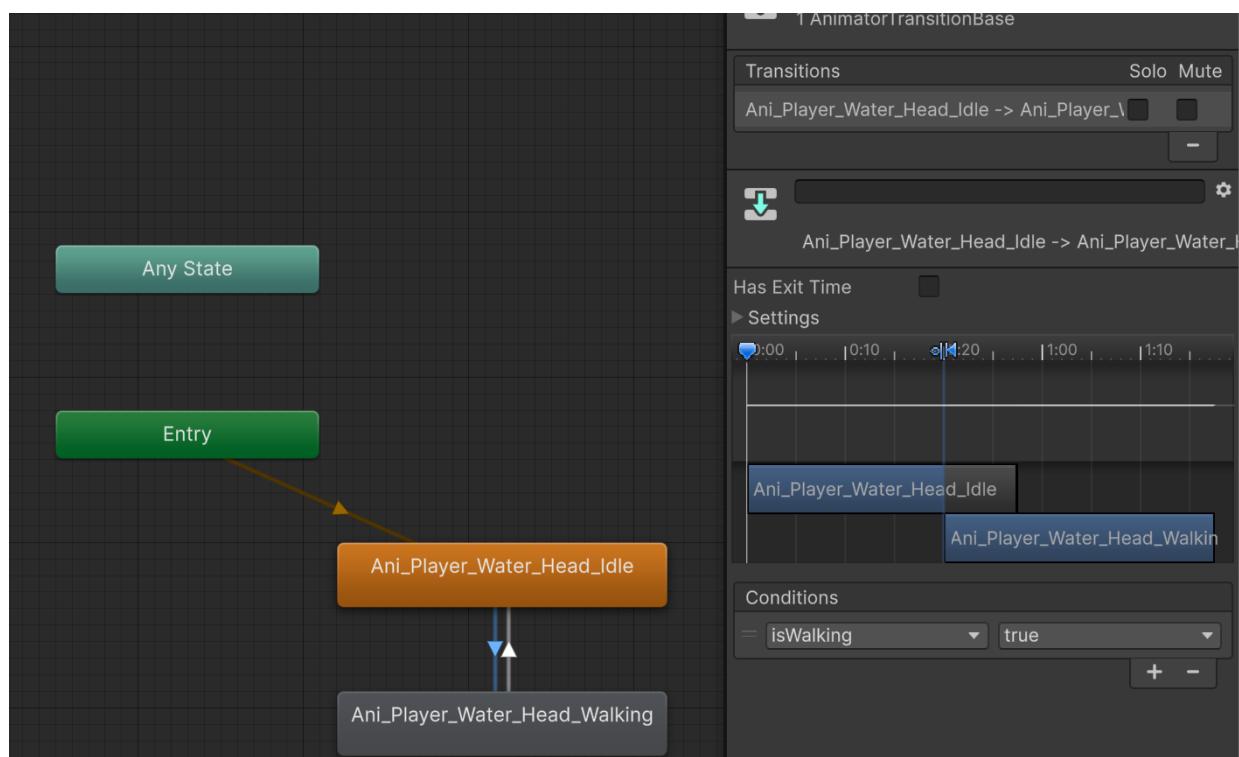
a. Ani_[Player]_[NomePersonagem]_[ParteDoCorpo]_[NomeAnimacao]

6. Veja os nomes das animações e tente fazer algo parecido

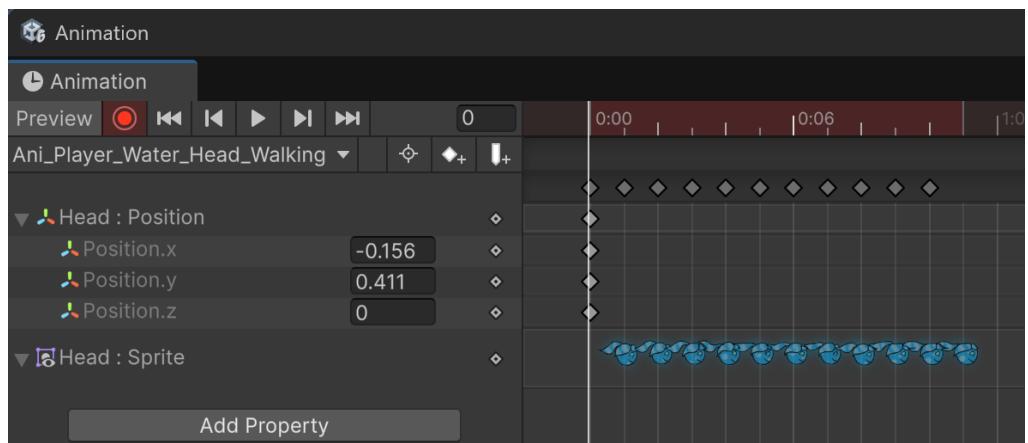


7. Configure transições entre os estados conforme os parâmetros.

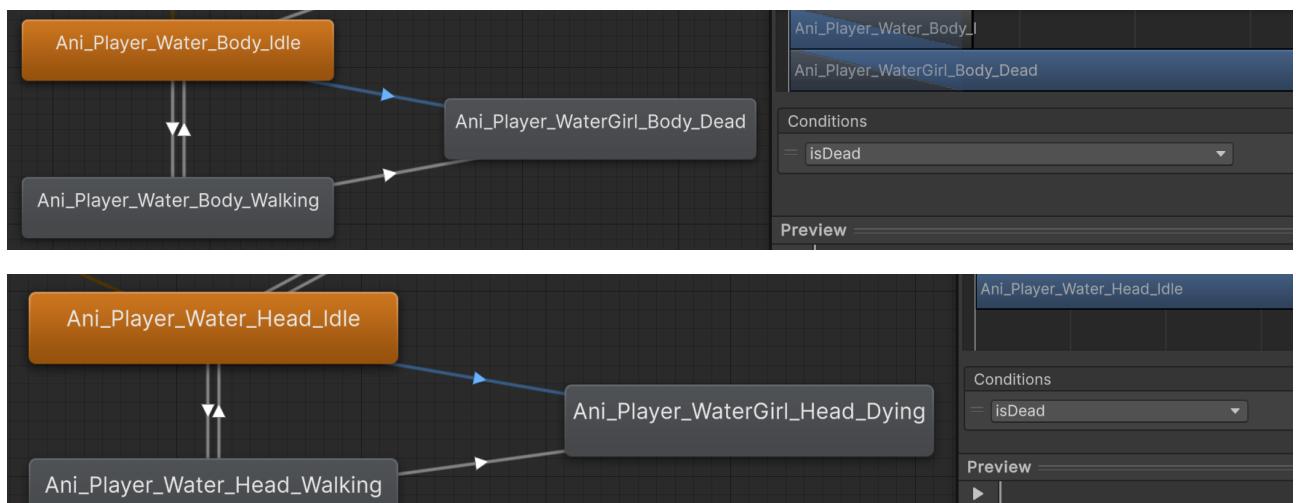
- Desative Has Exit Time para que não precise terminar a animação em execução antes de executar outra (onde necessário)
- Seta para baixo : Colocar isWalking para true
- Seta para cima: Colocar is Walking para false



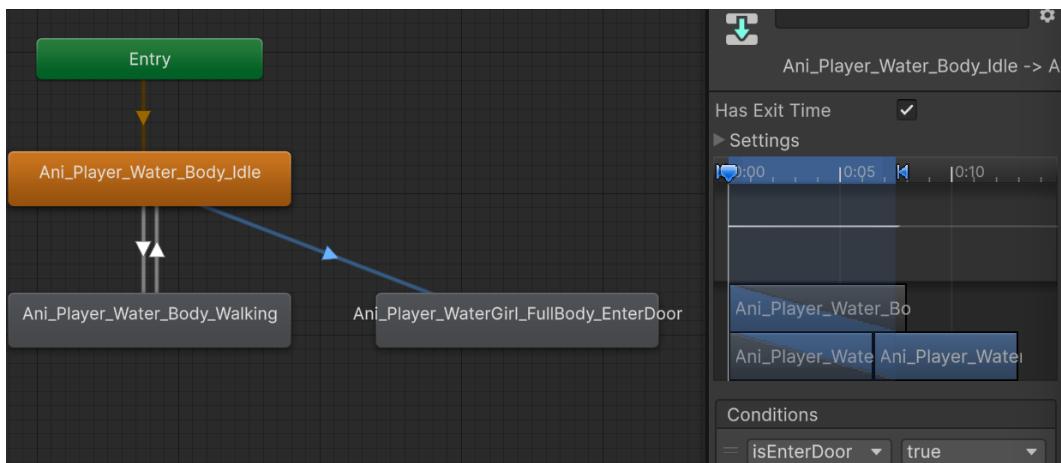
8. Dependendo da situação, ao fazer a movimentação do personagem a cabeça pode subir, e nesse caso você irá precisar animar para que ao animar a cabeça fique mais para baixo e pode até simular ela dando pequenos pulos ou se movendo.



9. Faça a parte da animação de morrer

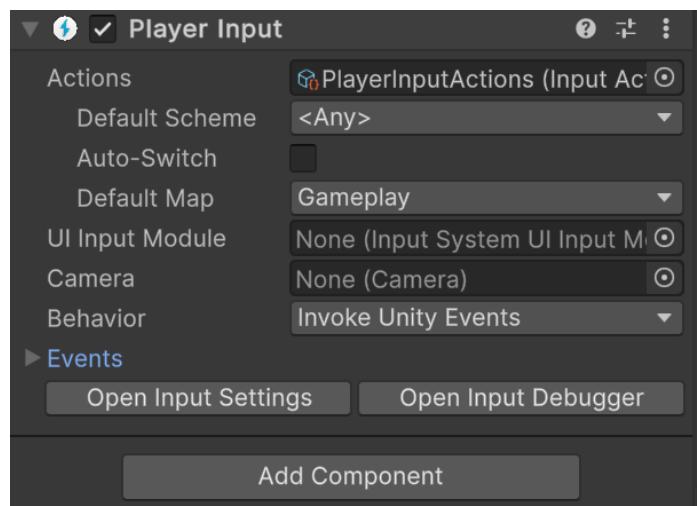


10. Faça a parte da animação de entrar na porta



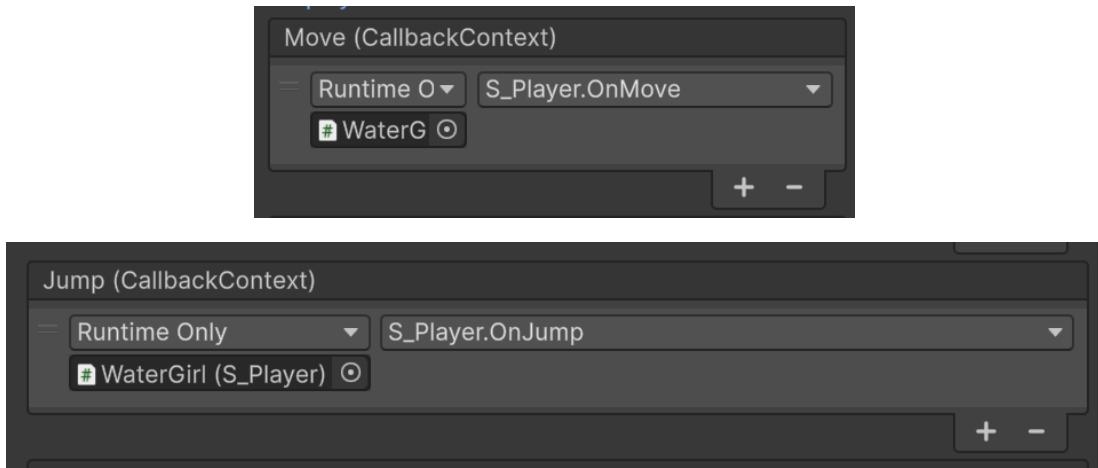
11. Adicione o *Player Input* ao seu Player. E configure como informado:

- a. Action Asset: *PlayerInputActions*
- b. Desmarque a opção Auto-Switch
- c. Default Map: *Gameplay*
- d. Behavior: Invoke Unity Events



12. Clique em Events -> Gameplay

13. Ache o CallbackContext denominado Move e Jump e arraste o WaterGirl e selecione as funções OnMove e OnJump.



Ao adicionar o componente PlayerInput a um GameObject na Unity, você verá um campo chamado Behavior, que define como as ações configuradas no InputActions serão executadas no seu código. Existem três opções disponíveis, e a escolha correta depende de como você deseja lidar com os comandos de entrada dentro do seu jogo.

A opção Send Messages é uma das mais utilizadas e a mais adequada para projetos onde cada personagem possui um script próprio com métodos específicos para movimentação, pulo, ataques, etc. Quando essa opção está ativa, a Unity procura automaticamente por métodos com nomes padronizados no script, como OnMove, OnJump, entre outros, e os chama quando a ação correspondente é executada. Esses métodos devem seguir a assinatura esperada pela Unity, como por exemplo:

```
public void OnMove(InputAction.CallbackContext context) { ... }
```

O principal benefício de usar o Send Messages é que não há necessidade de fazer ligações manuais de eventos nem arrastar referências no Inspector. É um modo direto e prático, ideal para jogos com múltiplos jogadores onde cada um é instanciado com seu próprio controle e lógica separada — exatamente como no jogo Foguinho e Aguinha. Ele também favorece a modularidade, pois cada jogador pode ter seu script com sua lógica individual, tornando o projeto mais organizado.

Já a opção Broadcast Messages funciona de maneira semelhante, mas com uma diferença importante: ela envia a mensagem para o GameObject principal e para todos os seus objetos filhos. Isso é útil quando você tem scripts que respondem às ações em objetos filhos, como uma cabeça ou corpo do personagem que possuem scripts separados. Porém, como envia a mensagem para vários objetos, pode impactar um pouco a performance em projetos maiores ou com muitos objetos aninhados.

Por fim, a opção Invoke Unity Events permite configurar manualmente as respostas para cada ação diretamente pelo Inspector. Com ela, você pode arrastar GameObjects e vincular métodos públicos diretamente a cada ação. Essa opção é útil em prototipagens rápidas ou para desenvolvedores que preferem configurar comportamentos de forma visual, sem escrever os métodos com nome específico. No entanto, ela exige que você faça todas as ligações manualmente, o que pode ser trabalhoso e mais propenso a erros em projetos maiores.

Sobre o Auto-Switch Auto-Switch habilitado permite que, ao apertar qualquer botão em um novo controle (teclado ou gamepad), a Unity associe automaticamente aquele dispositivo ao PlayerInput mais recente. Em contexto multiplayer: ON: funciona bem se cada jogador for criado com PlayerInput.Instantiate() e o dispositivo for emparelhado ali mesmo (como você já está fazendo no S_GameManager). OFF: requer que você defina manualmente o controle de cada jogador. Mais controle, mas mais código. Como você está criando os jogadores corretamente com pairWithDevice: device, o Auto-Switch pode permanecer ativado sem causar conflito — a não ser que veja comportamentos estranhos (tipo controle trocando de jogador no meio da partida, o que neste caso não está ocorrendo).

Criando o Script de Movimento e Animação

Os personagens devem responder a controles específicos e ter animações.

No script S_Player.cs, adicione:

```
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.InputSystem;

[RequireComponent(typeof(Rigidbody2D))]
public class S_Player : MonoBehaviour
{
    public GameObject head;
    public GameObject body;
    public float speed = 3f;
    public float jumpForce = 5f;
    public string characterType;

    private Rigidbody2D rb;
    private Animator animatorHead;
    private Animator animatorBody;
    private Vector2 moveInput;
    private bool jumpInput;
    private bool isGrounded;

    public AudioClip deathSound;
    private AudioSource audioSource;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        animatorHead = head.GetComponent<Animator>();
        animatorBody = body.GetComponent<Animator>();

        audioSource = GetComponent<
```

```

void Update()
{
    float move = moveInput.x * speed;
    rb.linearVelocity = new Vector2(move, rb.linearVelocity.y);

    animatorHead.SetBool("isWalking", Mathf.Abs(move) > 0.1f);
    animatorBody.SetBool("isWalking", Mathf.Abs(move) > 0.1f);

    if (move != 0)
        transform.localScale = new Vector3(Mathf.Sign(move), 1, 1);

    if (jumpInput && isGrounded)
    {
        rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
        isGrounded = false;
        jumpInput = false;
    }
}

public void OnMove(InputAction.CallbackContext context)
{
    moveInput = context.ReadValue<Vector2>();
}

public void OnJump(InputAction.CallbackContext context)
{
    if (context.performed)
        jumpInput = true;
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Ground"))
        isGrounded = true;
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if ((characterType == "FireMan" && collision.CompareTag("WaterPuddle")) ||
        (characterType == "WaterGirl" && collision.CompareTag("FirePuddle")) ||
        collision.CompareTag("AcidPuddle"))
    {
        animatorHead.SetTrigger("isDead");
        animatorBody.SetTrigger("isDead");

        if (deathSound != null && audioSource != null)
            audioSource.PlayOneShot(deathSound);

        // Desativa movimentação
        this.enabled = false;
        Destroy(gameObject, 0.5f);

        // Chamar painel de derrota
        Object.FindFirstObjectByType<S_GameUIManager>().ShowLoseScreen();
    }
}

public static class InputPreserveWorkaround
{

```

```

[RuntimeInitializeOnLoadMethod]
static void PreserveMethods()
{
    var dummy = new S_Player();
    dummy.OnMove(default);
    dummy.OnJump(default);
}
}

```

14. Associe o script ao WaterGirl e FireMan.

Sempre que qualquer jogador colidir com um obstáculo fatal (ex: água, fogo ou ácido incompatível), o jogo pausará e mostrará o painel de "Você Morreu" com os botões de Reiniciar ou Sair.

Gerenciador de Jogadores

```

using UnityEngine;
using UnityEngine.InputSystem;
using System.Collections.Generic;

public class S_GameManager : MonoBehaviour
{
    public GameObject[] playerPrefabs;
    public Transform[] spawnPoints;
    public GameObject hudPrefab;
    public Transform hudParent;
    public Color[] playerColors;
    public string[] playerNames = { "WaterGirl", "FireMan", "Earthling", "AirKid" };

    private int nextIndex = 0;
    private InputAction joinAction;
    private HashSet<InputDevice> joinedDevices = new HashSet<InputDevice>();

    //reinicializar os controles ou o joinAction na nova fase.
    void Awake()
    {
        nextIndex = 0;
        joinedDevices.Clear();

        // Garante que o tempo volta ao normal e controles reiniciam ao trocar de cena
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        Time.timeScale = 1f;
        // Reset de estado se precisar
        nextIndex = 0;
        joinedDevices.Clear();
    }

    private void OnEnable()
    {
        joinAction = new InputAction(binding: "<Gamepad>/start");
        joinAction.AddBinding("<Keyboard>/enter");
    }
}

```

```

        joinAction.performed += ctx => TryAddPlayer(ctx.control.device);
        joinAction.Enable();
    }

    private void OnDisable()
    {
        joinAction.Disable();
    }

    void TryAddPlayer(InputDevice device)
    {
        if (nextIndex >= playerPrefabs.Length ||
            nextIndex >= spawnPoints.Length ||
            nextIndex >= playerColors.Length ||
            nextIndex >= playerNames.Length)
        {
            Debug.LogError("Índice fora do intervalo. Verifique os arrays de prefabs, spawnPoints, cores e nomes.");
            return;
        }

        if (joinedDevices.Contains(device))
        {
            Debug.Log("Esse dispositivo já foi usado para instanciar um jogador.");
            return;
        }

        joinedDevices.Add(device);

        var playerInput = PlayerInput.Instantiate(
            playerPrefabs[nextIndex],
            playerIndex: nextIndex,
            controlScheme: null,
            splitScreenIndex: -1,
            pairWithDevice: device
        );

        playerInput.transform.position = spawnPoints[nextIndex].position;

        var custom = playerInput.GetComponent<S_PlayerCustomization>();
        if (custom != null)
        {
            custom.ApplyCustomization(playerNames[nextIndex],
playerColors[nextIndex]);
        }

        GameObject hud = Instantiate(hudPrefab, hudParent);
        var hudScript = hud.GetComponent<S_PlayerHUD>();
        if (hudScript != null)
            hudScript.SetPlayer(playerInput);

        nextIndex++;
    }
}

```

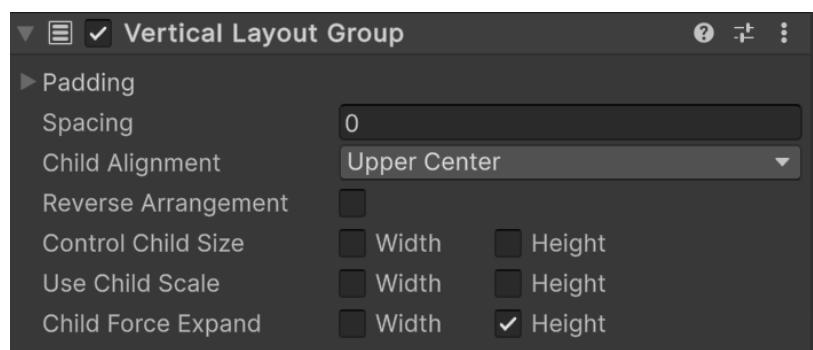
Criando o HUD do Personagem

O HUD será usado para mostrar o nome e a barra de vida do jogador. Ele será instanciado automaticamente para cada jogador com base em sua posição na lista de jogadores.

Configurar a HUD

Esse HUD usará o Canvas e componentes atuais

1. Crie um Canvas na cena através de `UI → Canvas` e renomeie para `Canvas_Player`
2. Certifique-se de que o Canvas está em modo Screen Space – Overlay
3. Ainda no Canvas, adicione um Canvas Scaler com as seguintes configurações:
 - a. UI Scale Mode: Scale With Screen Size
 - b. Reference Resolution: 1920 x 1080
 - c. Screen Match Mode: Match Width Or Height
 - d. Match: 0.5
4. Dentro do `Canvas_Player`, crie um Panel e renomeie como `HUD_Container`
5. No RectTransform, defina as seguintes âncoras e tamanho:
 - a. Alinhamento: Superior Direito
 - b. Anchor Min: (0, 1)
 - c. Anchor Max: (0, 1)
 - d. Pivot: (0, 1)
 - e. Position: X = -300, Y = -30
 - f. Width: 300, Height: 250
 - g. Layout: Adicione um componente Vertical Layout Group para organizar os HUDs empilhados verticalmente. E em Child Alignment deixe em *Upper Center*



- h. Ative Child Force Expand apenas para altura.

6. Dentro de *HUD_Container*, crie um objeto vazio chamado de *HUD_Player*
7. Dentro do *HUD_Player*, adicione:
 - a. TextMeshPro - Text e renomeie como *Txt_PlayerName*
 - b. Slider e renomeie como *Sli_HealthBar* (Configure o valor máximo como 100 e valor inicial como 100), desative o preenchimento e deixe o visual limpo, adicione uma cor personalizada no fill(conforme o personagem)
8. Crie um novo script *S_PlayerHUD.cs* e anexe ao *HUD_Player*

```
using UnityEngine;
using TMPro;
using UnityEngine.InputSystem;

public class S_PlayerHUD : MonoBehaviour
{
    public TextMeshProUGUI playerNameText;
    public UnityEngine.UI.Slider healthBar;

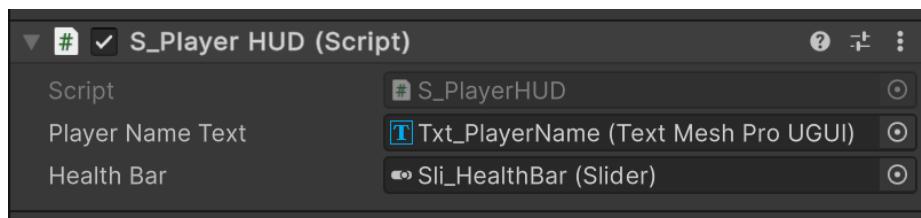
    public void SetPlayer(PlayerInput playerInput)
    {
        string[] nomes = { "WaterGirl", "FireMan", "EarthMan", "WindGirl" };
        int index = playerInput.playerIndex;

        if (playerNameText != null)
            playerNameText.text = nomes[index];

        // A barra de vida pode ser atualizada a partir de outro script (ex:
        S_PlayerHealth)
    }
}
```

9. Arraste *Txt_PlayerName* para Player name Text

10. Arraste *Sli_HealthBar* para Health Bar

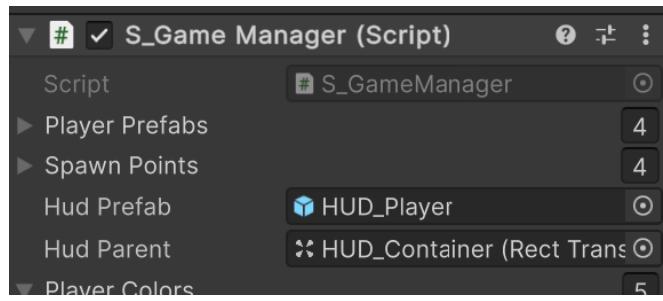


11. Transforme *HUD_Player* em Prefab arrastando para a pasta *Prefabs/HUD* e exclua da cena

No GameManager

Vamos agora configurar no *S_GameManager* para usar o HUD.

1. No campo *hudPrefab* arraste o prefab *HUD_Player*
2. No campo *hudParent* arraste o *HUD_Container* (o painel no Canvas)



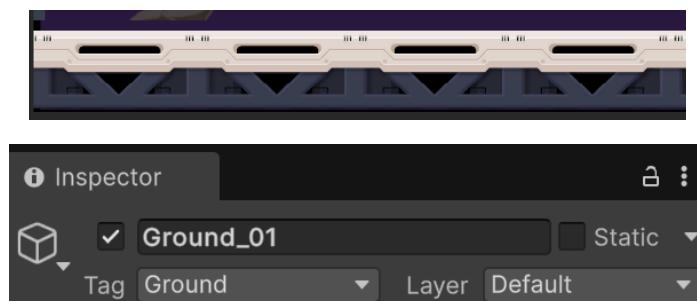
Criando o Cenário

Vamos agora criar a sua fase com bastante detalhe

Criar o chão

Será o chão do seu jogo

1. No Hierarchy, crie um novo GameObject com o sprite para chão, adicione um BoxCollider2D e renomeie para Ground_XX.
2. Crie a tag *Ground* e coloque em todo chão.



Criar Plataformas e Obstáculos

Será a plataforma e obstáculos

3. Use objetos 2D para criar plataformas.
4. Adicione BoxCollider2D para colisões.

Criando os Portais e Portas

Criação dos portais e portas do jogo

1. No Hierarchy, crie dois objetos vazios chamados Door_Fire e Door_Water.
2. Adicione BoxCollider2D e marque como "isTrigger".
3. No script S_FinalDoor, adicione:

```

using UnityEngine;

public class S_FinalDoor : MonoBehaviour
{
    public string requiredTag;
    public Sprite spriteFechada;
    public Sprite spriteAberta;

    private bool jogadorNaPorta = false;
    public bool JogadorNaPorta => jogadorNaPorta;

    private SpriteRenderer spriteRenderer;

    private void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        if (spriteRenderer != null && spriteFechada != null)
            spriteRenderer.sprite = spriteFechada;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag(requiredTag))
        {
            jogadorNaPorta = true;
            if (spriteRenderer != null && spriteAberta != null)
                spriteRenderer.sprite = spriteAberta;
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag(requiredTag))
        {
            jogadorNaPorta = false;
            if (spriteRenderer != null && spriteFechada != null)
                spriteRenderer.sprite = spriteFechada;
        }
    }
}

```

4. Associe o script às portas e configure requiredTag conforme necessário.
5. Defina a requiredTag como "FireMan" ou "WaterGirl" (você pode definir essas tags manualmente no Unity, clicando com botão direito no GameObject > Tag > Add Tag)
6. Defina os Sprite da porta Fechada e Aberta
7. Crie outro script S_DoorManager.cs para verificar quando todas estão ocupadas:

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class S_DoorManager : MonoBehaviour
{
    public S_FinalDoor[] portas;

    void Update()
    {

```

```

        foreach (var porta in portas)
    {
        if (!porta.JogadorNaPorta) return;
    }
    SceneManager.LoadScene("TelaDeVitoria");
}

```

8. Crie um objeto vazio chamado DoorManager
9. Associe esse script nele e preencha o array portas com Door_Fire e Door_Water

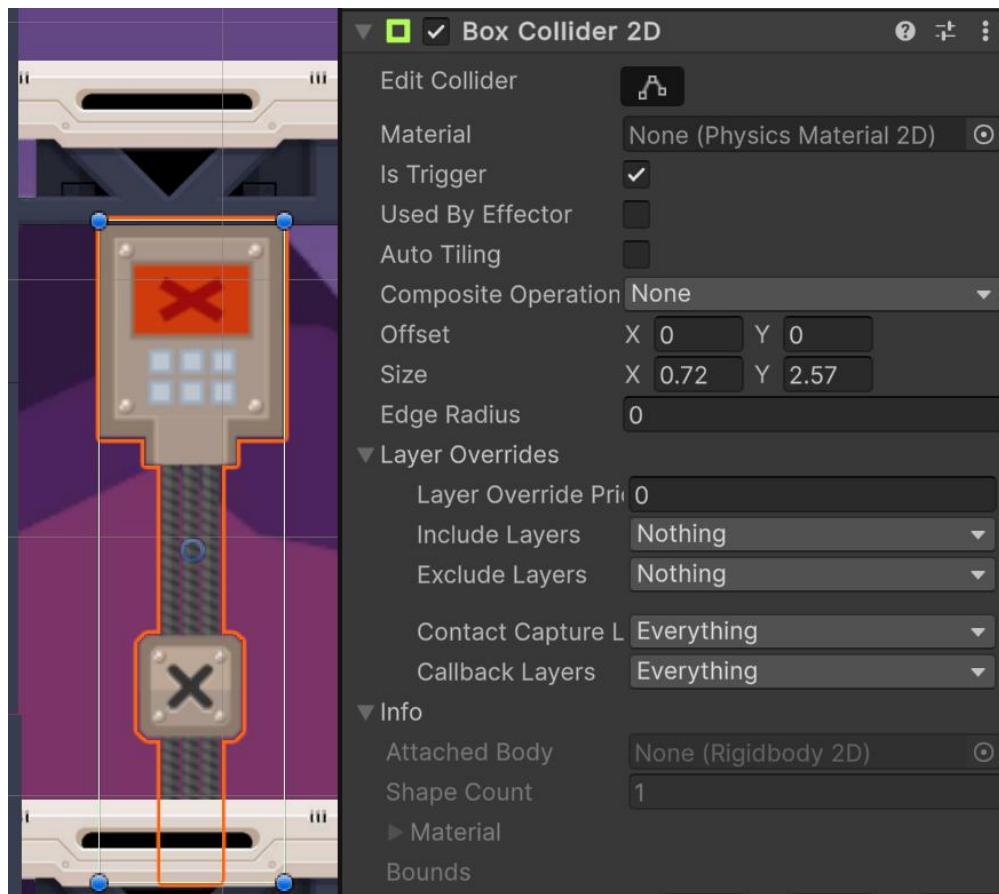
Criando Ativadores para Portões

Neste passo, vamos permitir que portões sejam abertos ao tocar em algum ativador. Isso adiciona uma mecânica cooperativa e interativa ao jogo.

Criando o Ativador

Iremos criar o ativador que será usado para ativar as portas para os elementos entrar.

1. Crie um objeto vazio chamado *ActivateGate*
2. Adicione o componente Sprite Renderer e um sprite que identifique um ativador
3. Adicione um componente BoxCollider2D
4. Marque a opção *Is Trigger*



5. Digite o script e anexe ao ActivateGate

```
using UnityEngine;

public class S_Gates : MonoBehaviour
{
    public GameObject gate;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("FireMan") || collision.CompareTag("WaterGirl") ||
            collision.CompareTag("Earth") || collision.CompareTag("Air"))
        {
            gate.SetActive(false); // Abre o portão
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("FireMan") || collision.CompareTag("WaterGirl") ||
            collision.CompareTag("Earth") || collision.CompareTag("Air"))
        {
            gate.SetActive(true); // Fecha novamente se quiser um botão temporário
        }
    }
}
```

Se quiser que o botão funcione como um botão permanente (abre uma vez e mantém aberto), remova o OnTriggerExit2D.

Criando o Portão que bloqueia um caminho

Iremos criar o portão que bloqueia um caminho

1. Crie um objeto chamado *GateBlock*
2. Adicione um SpriteRenderer e um BoxCollider2D
3. Certifique-se que o portão *GateBlock* bloqueia fisicamente alguma passagem
4. Arraste o *GateBlock* para o GameObject *ActivateGate* em Gate em S_Gates



5. Certifique-se que os jogadores possuem as Tags apropriadas: "FireMan", "WaterGirl", "Earth" e "Air"
6. Execute e veja o resultado

Permitindo alterar o Sprite no Portão quando ativado e desativado

Iremos alterar o código permitindo que o sprite seja trocado ao ser ativado e desativado, deixando o jogo bem mais interessante.

7. Altere o Código

```
using UnityEngine;

public class S_Gates : MonoBehaviour
{
    public GameObject gate;
    public Sprite defaultSprite;
    public Sprite pressedSprite;

    private SpriteRenderer spriteRenderer;

    private void Awake()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        if (spriteRenderer == null)
            Debug.LogWarning("S_Gates: Nenhum SpriteRenderer encontrado no objeto.");
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("FireMan") || collision.CompareTag("WaterGirl"))
        {
            gate.SetActive(false);
            if (spriteRenderer != null && pressedSprite != null)
```

```

        spriteRenderer.sprite = pressedSprite;
    }

}

private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("FireMan") || collision.CompareTag("WaterGirl"))
    {
        gate.SetActive(true);
        if (spriteRenderer != null && defaultSprite != null)
            spriteRenderer.sprite = defaultSprite;
    }
}

}

```

8. Selecione o Activate Gate e no S_Gates, configure Default Sprite colocando o sprite que estará quando desligado e em Pressed Sprite coloque o sprite que deve ficar ao ser ativado.



Interações com Lava, Água e Ácido e Áudio de Morte

Neste passo, vamos configurar os comportamentos fatais de cada personagem ao tocar em certos elementos perigosos do ambiente. Ao morrer uma animação será ativada de morte com um som e ele irá desaparecer.

Tags necessárias e configurações

1. Crie as seguintes tags no Unity, caso ainda não existam: *WaterPuddle*, *FirePuddle*, *AcidPuddle*, *EarthPuddle*, *AirPuddle*.
2. Adicione essas tags aos elementos correspondentes no seu cenário. A poça de água coloque *WaterPuddle* e sucessivamente.
3. Modifique Order in Layer = 2

Adicionando lógica no S_Player.cs

1. Inclua o código

```

using UnityEngine;

public AudioClip deathSound;
private AudioSource audioSource;

void Start()
{
    // ... (demais inicializações)
    audioSource = GetComponent<AudioSource>();
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if ((characterType == "FireMan" && collision.CompareTag("WaterPuddle")) ||
        (characterType == "WaterGirl" && collision.CompareTag("FirePuddle")) ||
        collision.CompareTag("AcidPuddle"))
    {
        animatorHead.SetTrigger("isDead");
        animatorBody.SetTrigger("isDead");

        if (deathSound != null && audioSource != null)
            audioSource.PlayOneShot(deathSound);

        this.enabled = false;
        Destroy(gameObject, 0.5f);
    }
}

```

Com esse código a lógica seria que um personagem de fogo morre se tocar em água. Um personagem de água morre se tocar em lava. Todos morrem se tocarem em ácido.

Criando o Sistema de Reinício e Vitória

Agora vamos criar um sistema que detecta quando o jogador morreu ou venceu, e exibe um menu apropriado. Vamos criar duas telas diferentes para exibir ao final do jogo: uma de vitória e outra de derrota, com opções como "Reiniciar", "Sair" e "Próxima Fase".

Criando o UI de Fim de Jogo

1. Crie um Canvas na cena através de **UI → Canvas** e renomeei para MainCanvas
2. Certifique-se de que o Canvas está em modo Screen Space – Camera
3. Em Render Camera arraste MainCamera
4. Ainda no Canvas, adicione um Canvas Scaler com as seguintes configurações:
 - a. UI Scale Mode: Scale With Screen Size
 - b. Reference Resolution: 1920 x 1080
 - c. Screen Match Mode: Match Width Or Height
 - d. Match: 0.5

5. No Canvas, crie dois Panels
 - a. Panel_Win, visível somente em caso de vitória
 - b. Panel_Lose, visível somente em caso de morte
6. Modifique para que ambos tenham Anchor Preset: center-middle e tamanho 500 x 500
7. Dentro de cada painel, adicione botões com os seguintes nomes e textos:
 - a. Text (TMP): mensagem (ex: "Parabéns!", "Você Morreu")
 - b. Button (TMP): Btn_Restart, renomeei o texto para Restart
 - c. Button (TMP): Btn_NextLevel (somente no painel de vitória), renomeei o texto para Next
 - d. Button (TMP): Btn_Quit , renomeei o texto para Quit
8. Faça os painéis como preferir e deixe bem bonito
9. Desative os dois painéis no início da cena

Script de controle S_GameUIManager.cs

1. Digite o script

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class S_GameUIManager: MonoBehaviour
{
    public GameObject winPanel;
    public GameObject losePanel;

    public void ShowWinScreen()
    {
        winPanel.SetActive(true);
        Time.timeScale = 0f;
    }

    public void ShowLoseScreen()
    {
        losePanel.SetActive(true);
        Time.timeScale = 0f;
    }

    public void Restart()
    {
        Time.timeScale = 1f;
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void Quit()
    {
        Application.Quit();
    }
}
```

```

        public void NextLevel(string nextLevel)
{
    Time.timeScale = 1f;
    SceneManager.LoadScene(nextLevel);
}
}

```

Exibindo o Menu de Fim de Jogo

Para ativar os menus de vitória ou derrota em tempo real.

1. No script do jogador S_Player, adicione no momento da morte:

```
FindObjectOfType<S_GameUIManager>().ShowLoseScreen();
```

Esse método é chamado quando o personagem morre (após animação de morte ou colisão com elemento nocivo). Se preferir mais robustez, use um GameManager central para isso e envie um evento para ele.

2. No final do método Update() em S_DoorManager.cs, substitua:

```
//SceneManager.LoadScene("TelaDeVitoria");
//por
Object.FindFirstObjectOfType<S_GameUIManager>().ShowWinScreen();
```

Esse método é chamado após a verificação de que todos os jogadores chegaram às suas portas. É recomendado garantir que a função só seja chamada uma vez (pode usar bool vitoriaMostrada para controlar isso).

Agora, ao morrer ou vencer, o jogo pausa e exibe o menu apropriado com opções para reiniciar, sair ou ir para a próxima fase!

Configurando os botões no Inspector (OnClick)

Para todos os botões criados, configure o campo On Click() assim:

3. No botão Btn_Restart faça (nos dois painéis):
 - a. OnClick → +
 - b. Arraste o objeto com o script S_GameOverUI
 - c. Escolha a função: S_GameUIManager → Restart()
4. No botão Btn_Close (nos dois painéis):
 - a. OnClick → +

- b. Objeto: S_GameUIManager
 - c. Função: S_GameUIManager → Quit()
- 5.** No Btn_NextLevel (apenas no painel de vitória):
- a. OnClick → +
 - b. Objeto: S_GameUIManager
 - c. Função: S_GameUIManager → NextLevel()
- 6.** Certifique-se de que o script S_GameUIManager está presente na cena e os objetos winPanel e losePanel foram atribuídos corretamente no Inspector.

Tela de Seleção de Fase

Agora vamos criar uma tela onde o jogador poderá escolher a fase que deseja iniciar.

Criando a cena de seleção de fase

- 1.** Crie uma nova cena chamada SCN_LevelSelect
- 2.** No Canvas, adicione um Panel e renomeie para Panel_Levels
- 3.** Dentro do Panel, crie botões para cada fase disponível:
 - a. Btn_Level1, texto Level 1
 - b. Btn_Level2, texto Level 2
 - c. Level 3 (e assim por diante)
- 4.** Para cada botão, adicione um evento no OnClick() chamando um método que carrega a fase correspondente.

Script S_SelecionarFase.cs

- 1.** Crie e digite o código

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class S_LevelSelect : MonoBehaviour
{
    public void LoadLevel(string levelName)
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(levelName);
    }
}
```

}

Configurando os botões

- 1.** Crie um GameObject vazio chamado S_LevelSelect
- 2.** Anexe o script S_LevelSelect.cs
- 3.** Nos botões da UI:
 - a. Arraste o LevelSelect para o campo de evento OnClick()
 - b. Escolha S_LevelSelect -> LoadLevel(string)
 - c. Preencha o nome exato da cena (ex: Level1)
- 4.** Lembre-se de adicionar todas essas cenas no Build Settings (File > Build Settings) para que possam ser carregadas corretamente.