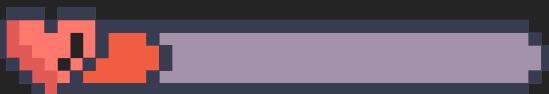
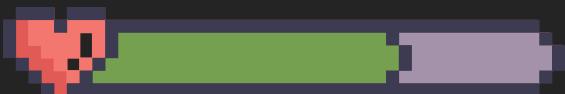


2025 EXPERIMENTA-TE PROGRAM

# FIGHTER GAME TUTORIAL

JIQI WANG  
JOÁO VITOR

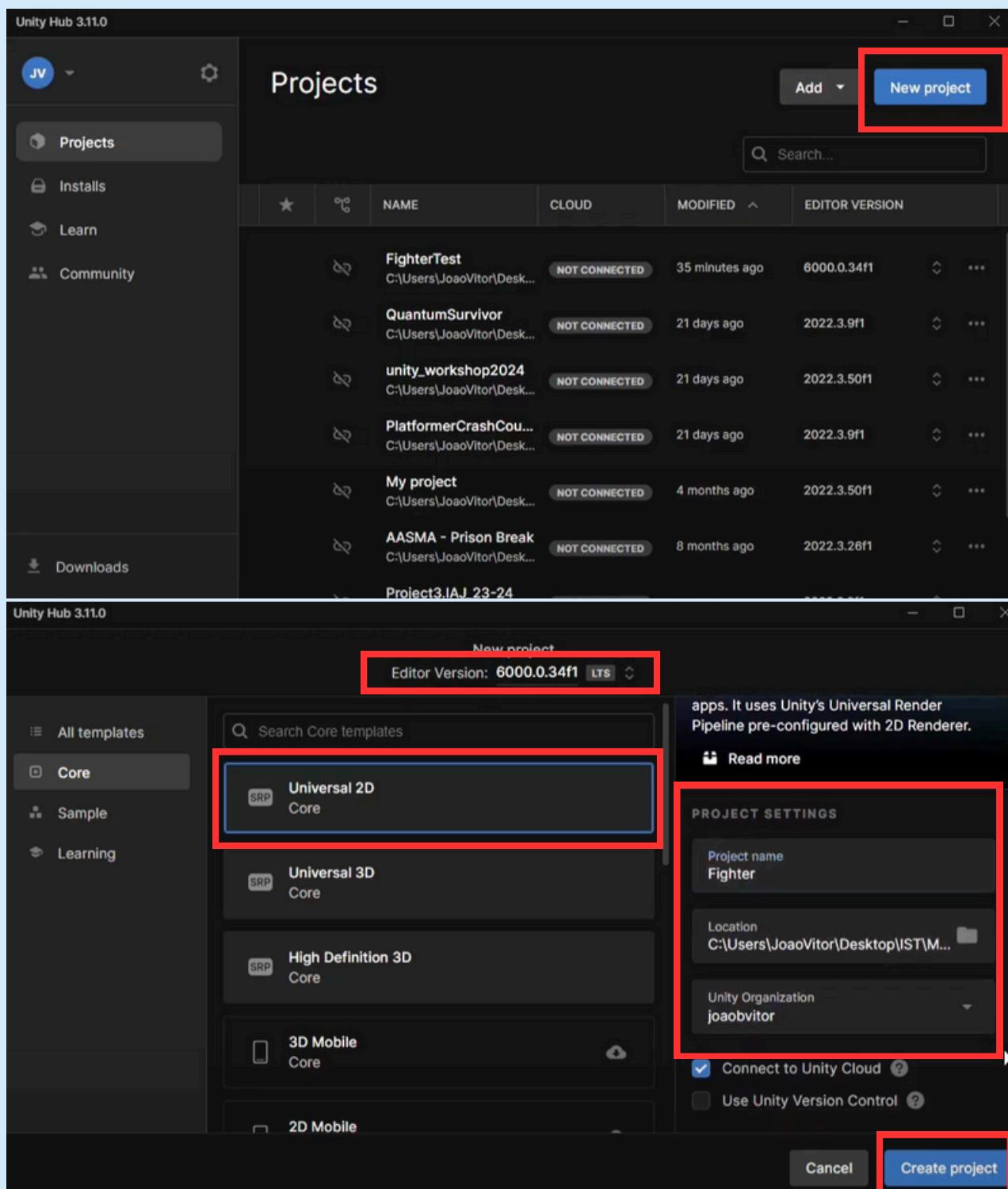
START

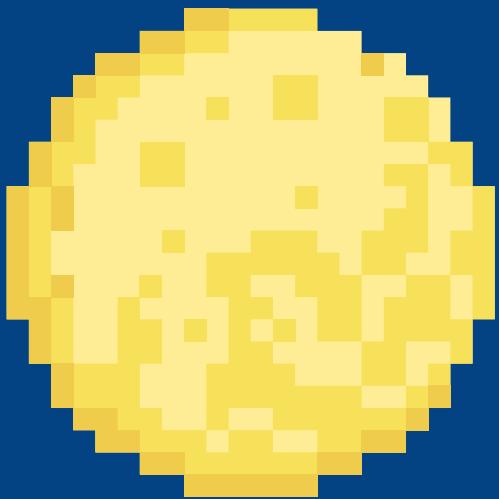


# FIGHTER GAME TUTORIAL

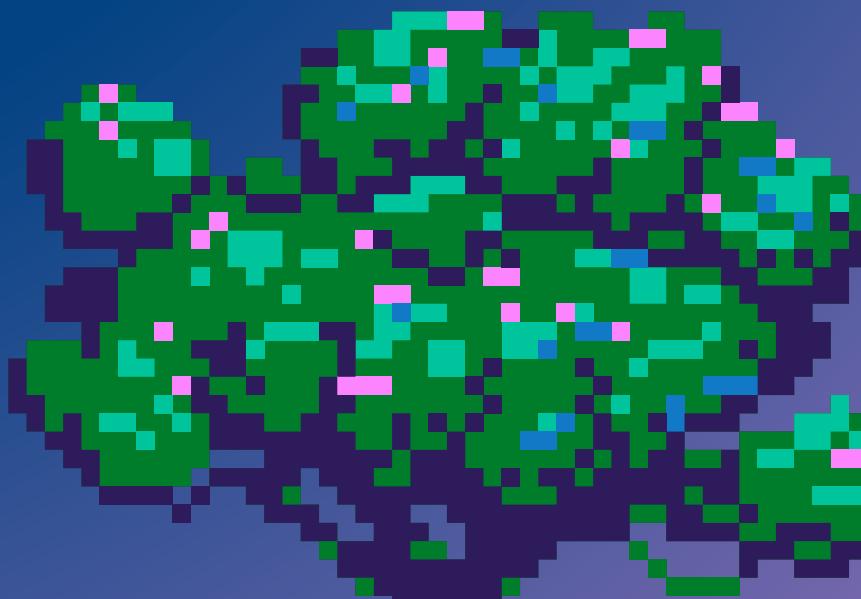
## Setup de novo projeto em Unity

- 1- Ter Unity instalado. (<https://unity.com/download>)
- 2- Criar conta e/ou entrar com conta Unity.
- 3- Clicar “New Project”, usar a versão de Unity mais recente, escolher “Universal2D”, decidir o nome do jogo e uma localização. No final, clicar em “Create Project”.



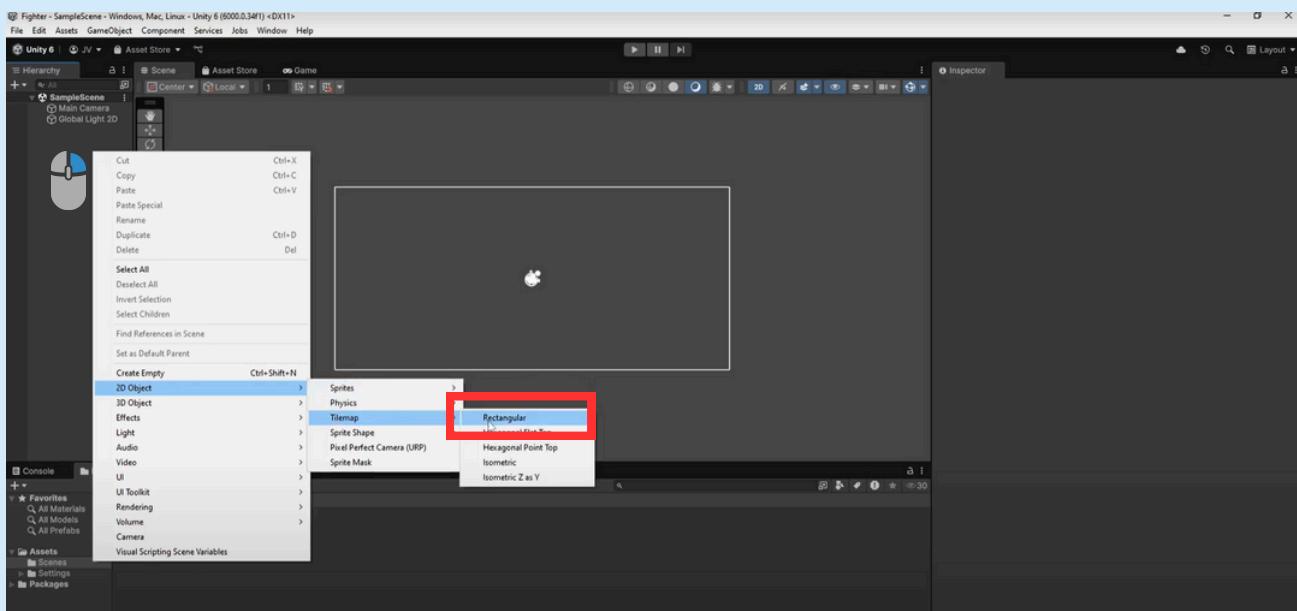


TI-EMAP

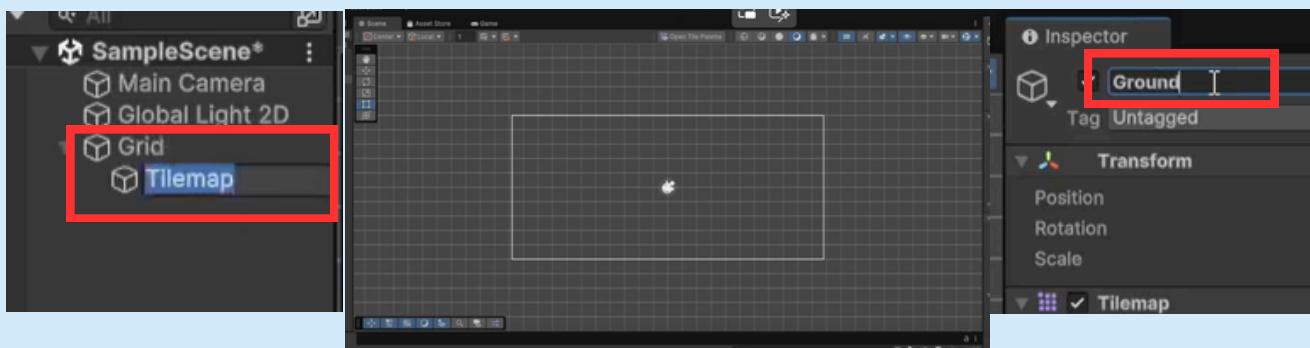


## Criação de Mapa, aplicação de Assets

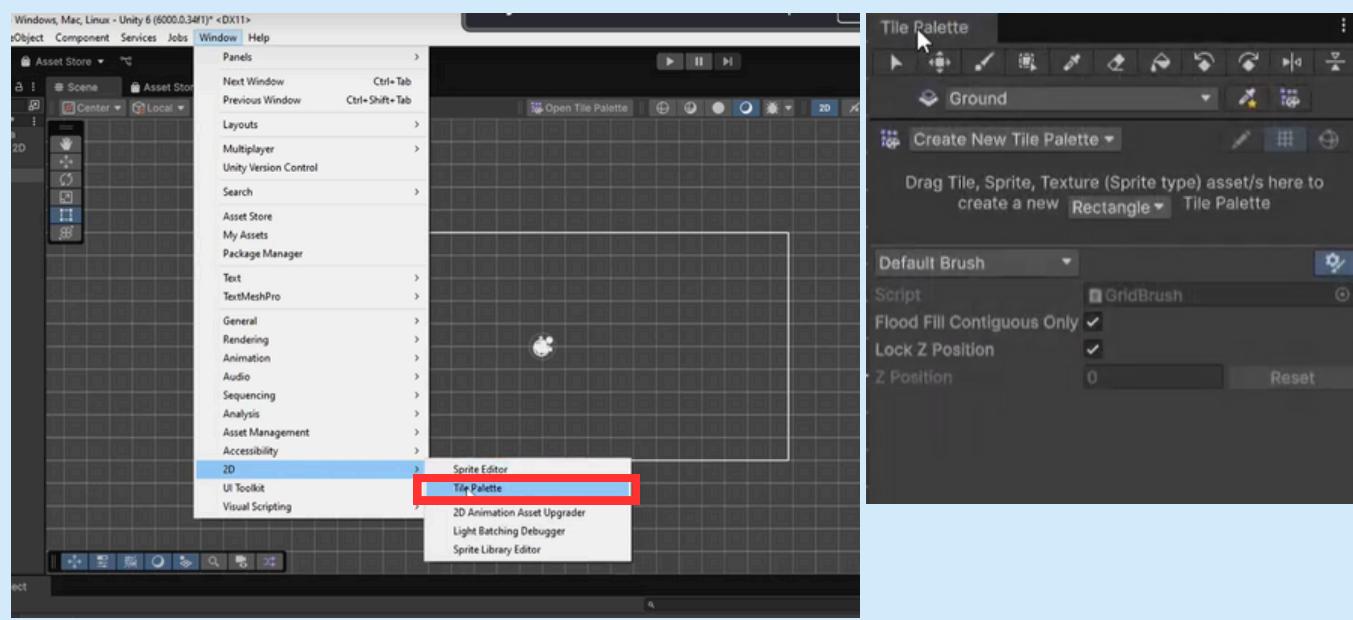
1- Para criar o mapa inicial, dentro da SampleScene, usar o botão direito do rato encontrar “2D Object”→“Tilemap”→“Retangular” e clicar.



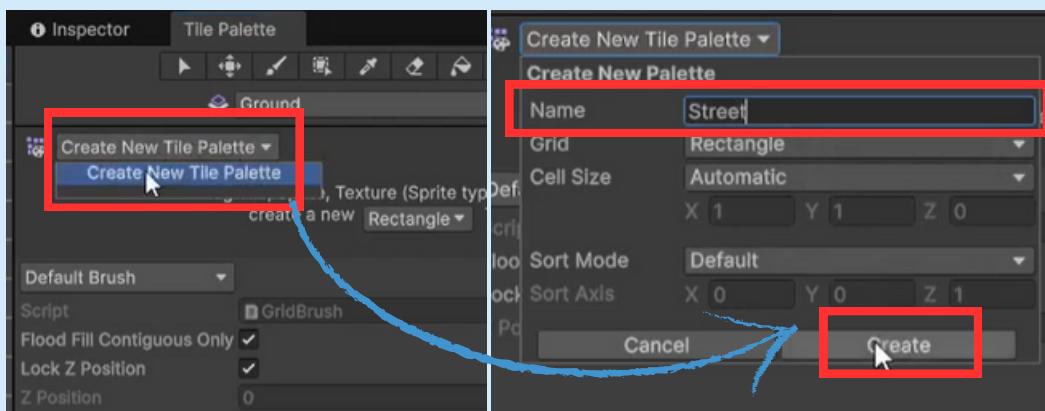
2- A Scene terá o seguinte aspecto e irá aparecer no lado esquerdo uma Grid com Tilemap. Podem alterar o nome para Ground em vez de Tilemap.



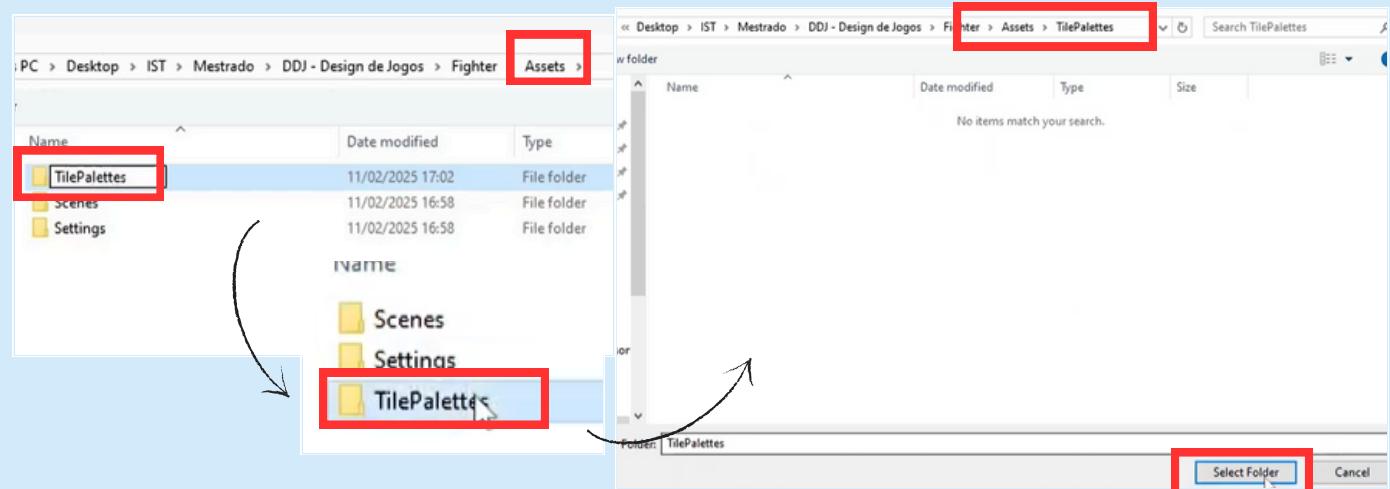
3- Para desenhar o mapa, procuramos a tab de “Windows”→“2D”→“Tile Palette” e clicamos. Isto vai abrir uma nova tab chamada Tile Palette, e é com isso que desenhemos o nosso mapa.



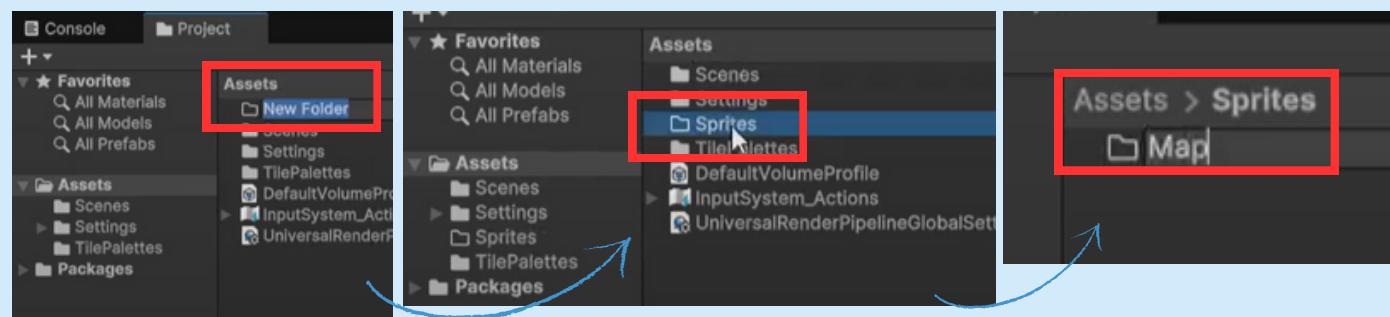
4- Clicamos em “Create New Tile Palette”, mudamos o nome e clicamos em “Create”.



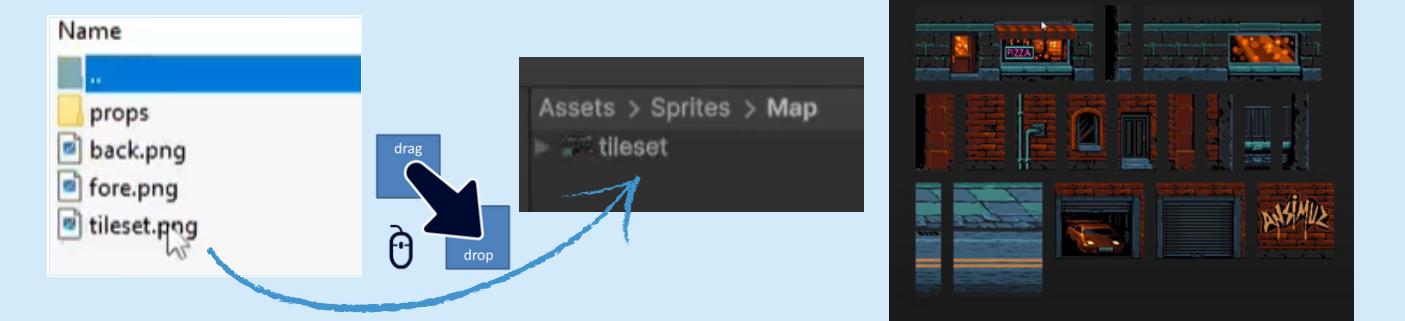
5- Ao criar, irá pedir a pasta para onde guardar. Para isso, temos de criar uma pasta chamada “TilePalettes” dentro da pasta de “Assets”. No fim, selecionar esta pasta criada.



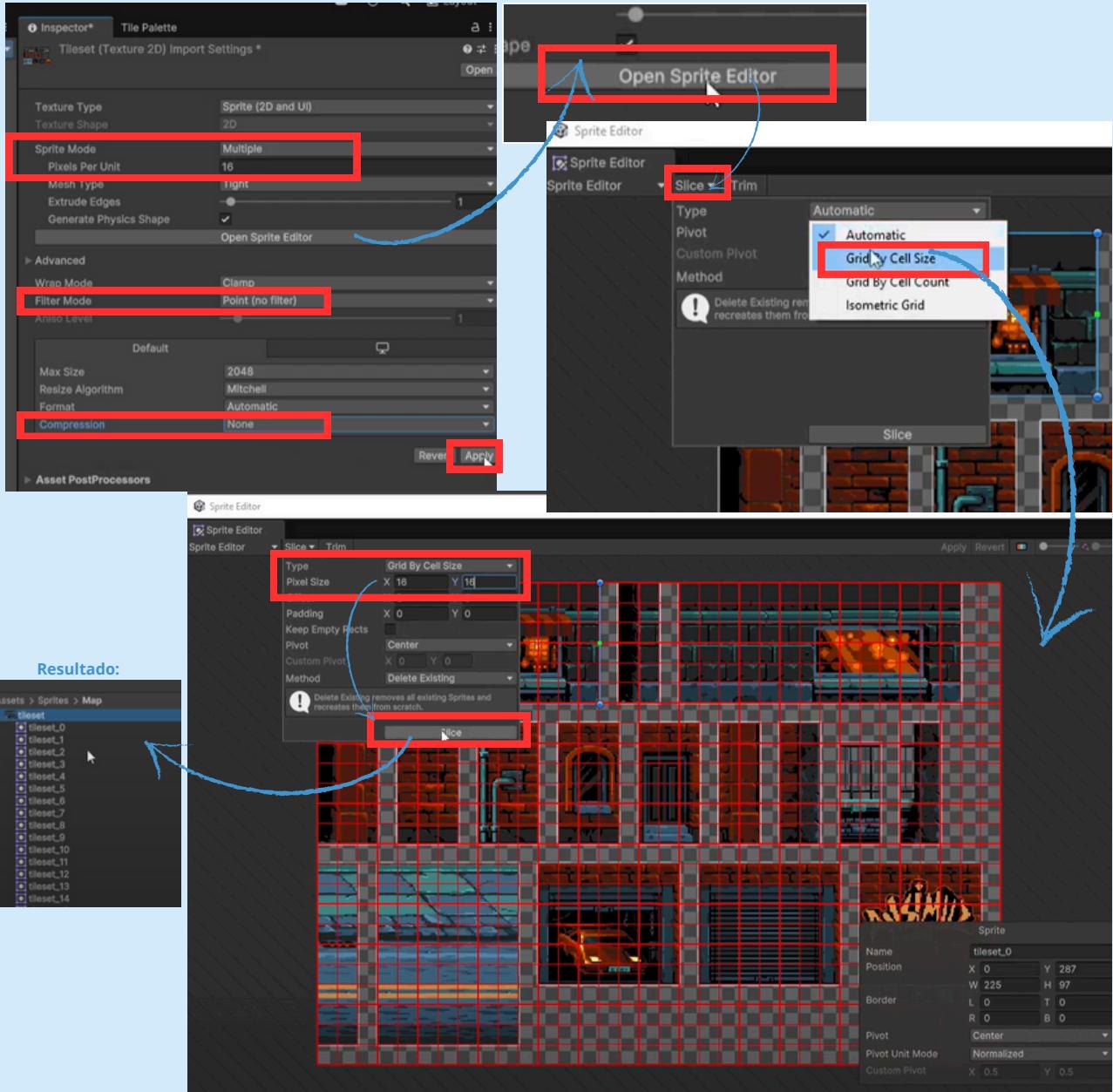
6- Para usarmos os Assets do chão, precisamos de ir à pasta de Assets e criar uma pasta chamada “Sprites”. Dentro desta pasta, criamos outra pasta dedicada ao mapa.



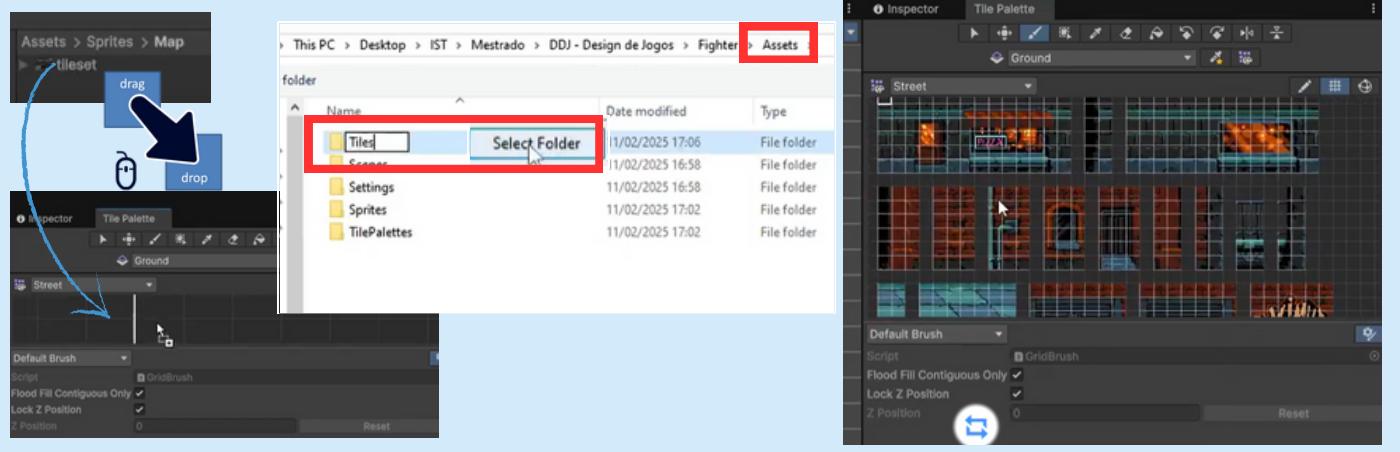
7- Dentro do zip de Assets Streets of Fight (<https://ansimuz.itch.io/streets-of-fight>), entramos na pasta de “Stage Layers” e arrastamos a imagem de tileset para dentro do nosso Unity, na pasta de Map.



8- Para recortarmos os Sprites, temos de garantir os seguintes aspectos da tileset e aplicar qualquer mudança. Após isso, abrimos o Sprite Editor, selecionamos “Slice” e alterarmos o Type para “Grid by Cell Size”. No fim, usamos um Pixel Size de 16 por 16 e aplicamos as mudanças.



9- Arrastando o tileset para dentro do Tile Palette, irá pedir a localização para guardarmos as imagens recortadas. Criamos uma pasta “Tiles” dentro de Assets e iremos ter o nosso mapa.

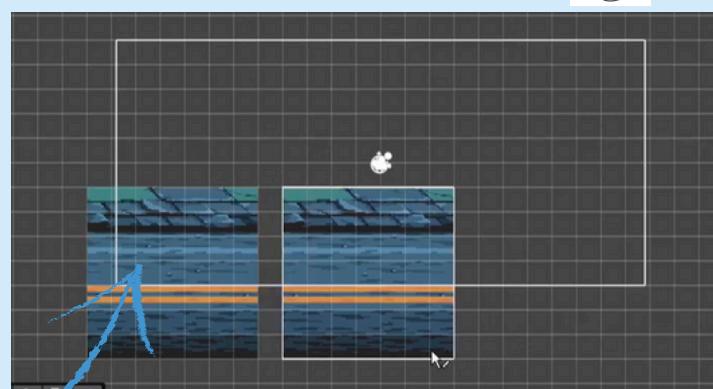
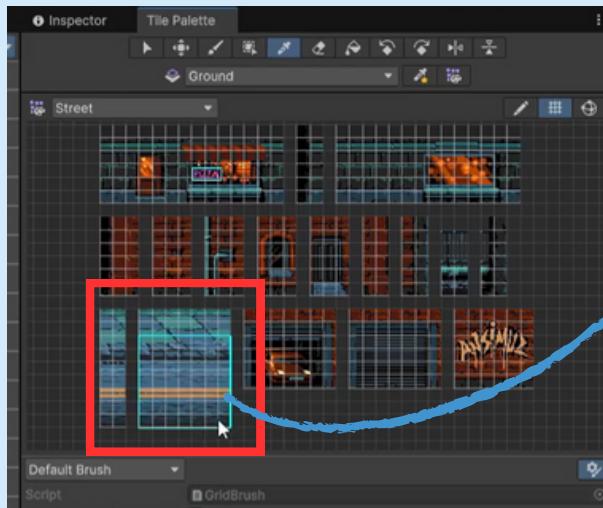


10- Para desenharmos o chão, clicamos e arrastamos com o rato por cima da área

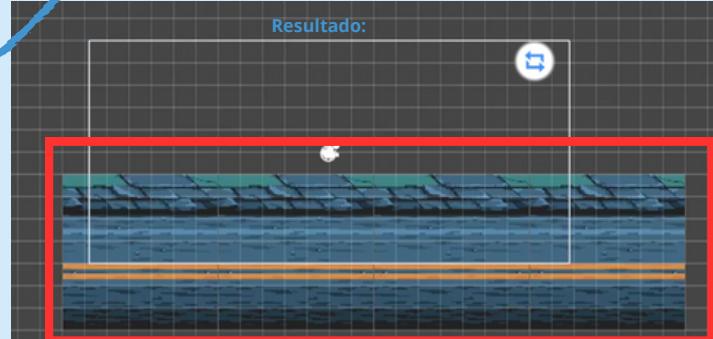


que queremos, e desenharmos na Scene

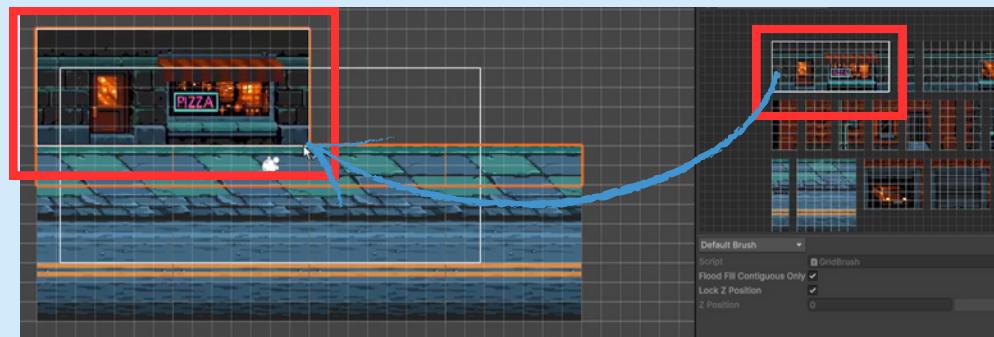
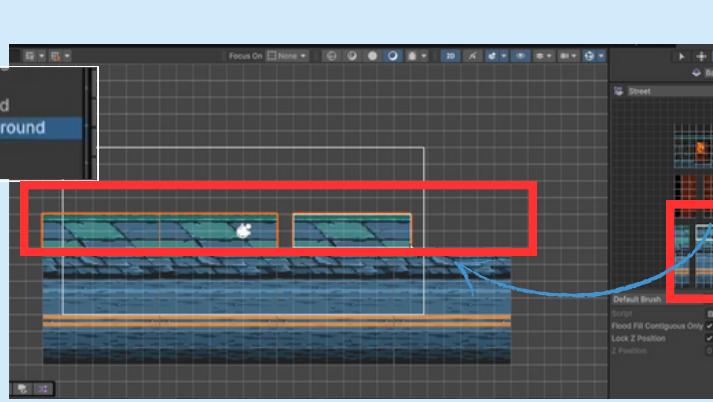
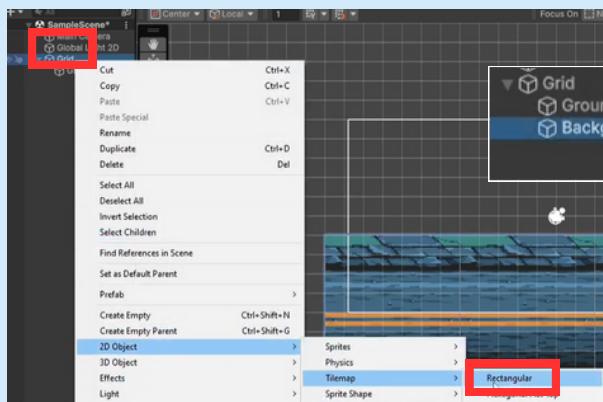
clicando onde queremos pintar.



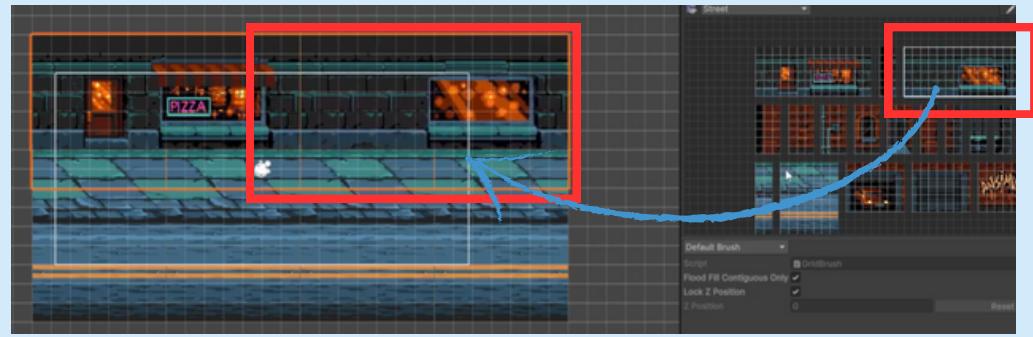
Resultado:



11- Para desenharmos o background, dentro de Grid, criamos outro Rectangular Tilemap chamado “Background”, e repetimos as etapas acima para desenhar.



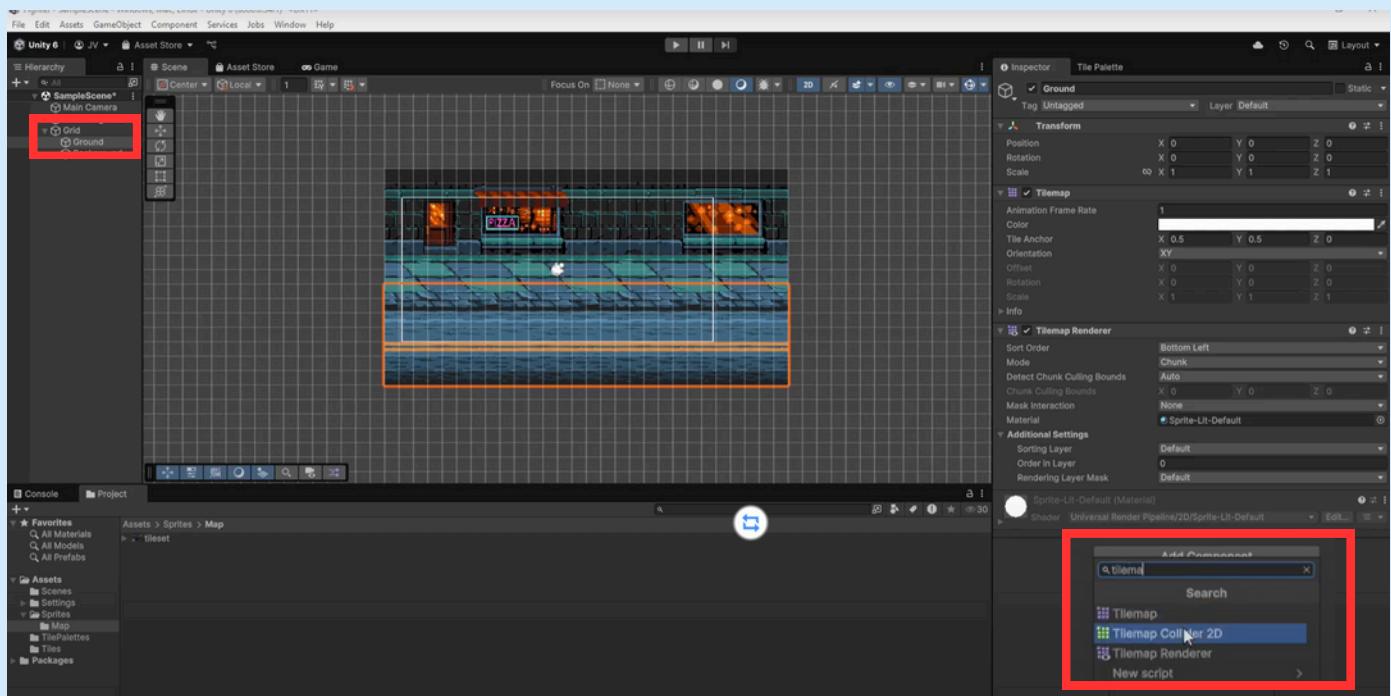
Default Brush  
Script: GridBrush  
Flood Fill Contiguous Only  
Lock Z Position  
Z Position: 0



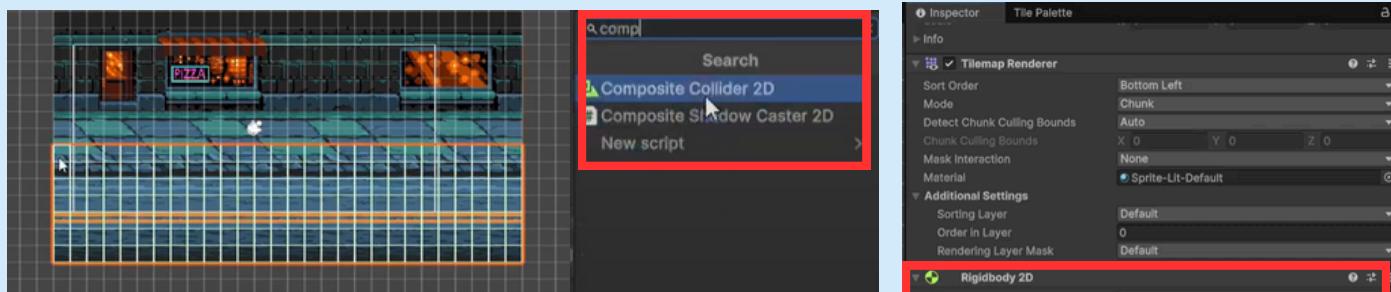
Default Brush  
Script: GridBrush  
Flood Fill Contiguous Only  
Lock Z Position  
Z Position: 0

# Criação de Mapa, aplicação de Colliders em plataformas

1- Para o jogador puder andar no chão e não cair para baixo, temos de aplicar um collider ao Ground. Para isso, selecionamos o Ground, adicionamos uma componente chamada “Tilemap Collider 2D”.

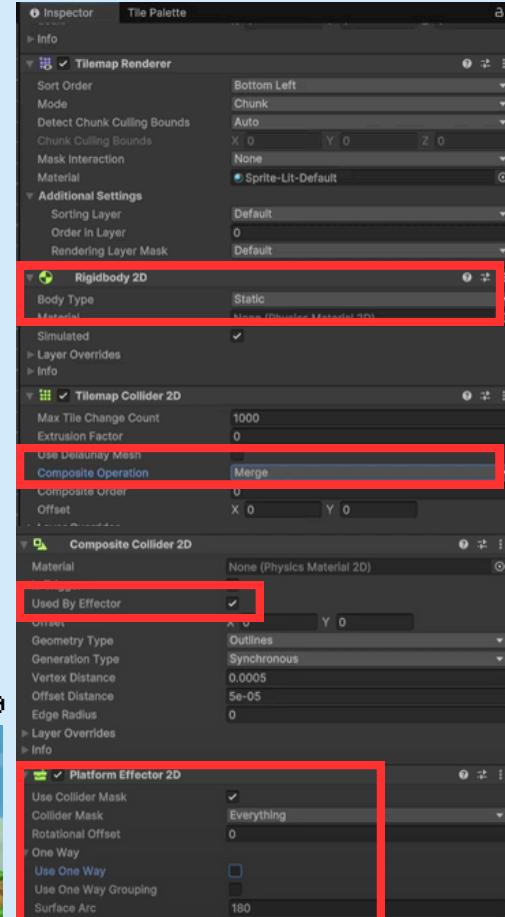


2- Depois de aplicarmos o collider, conseguimos ver que cada um dos quadrados do chão tem um collider. Para juntar os quadrados como um só, adicionamos outra componente chamada “Composite Collider 2D” e alteramos a Composite Operation para Merge.



Ao adicionar o Composite Collider, cria automaticamente uma “Rigidbody 2D”. Esta componente serve para aplicar físicas. Como não queremos fazer alterações sobre o chão, alteramos a Body Type para “Static”.

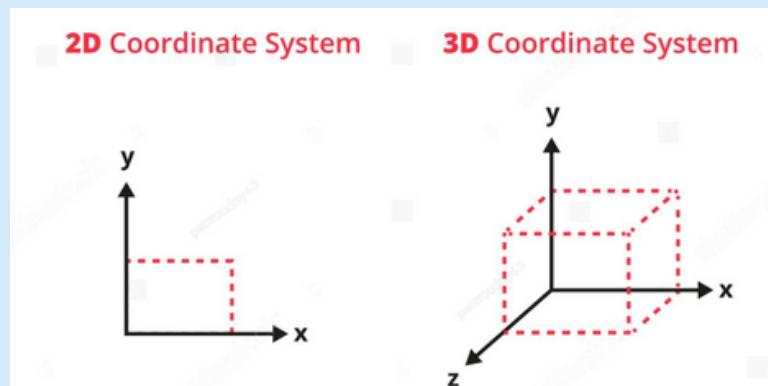
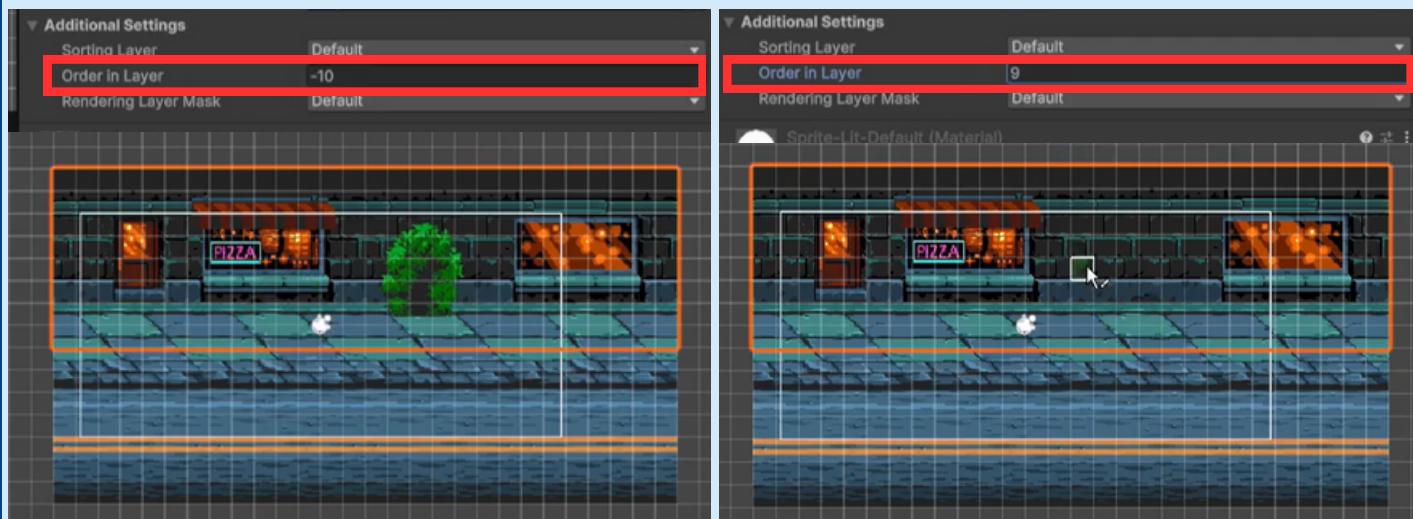
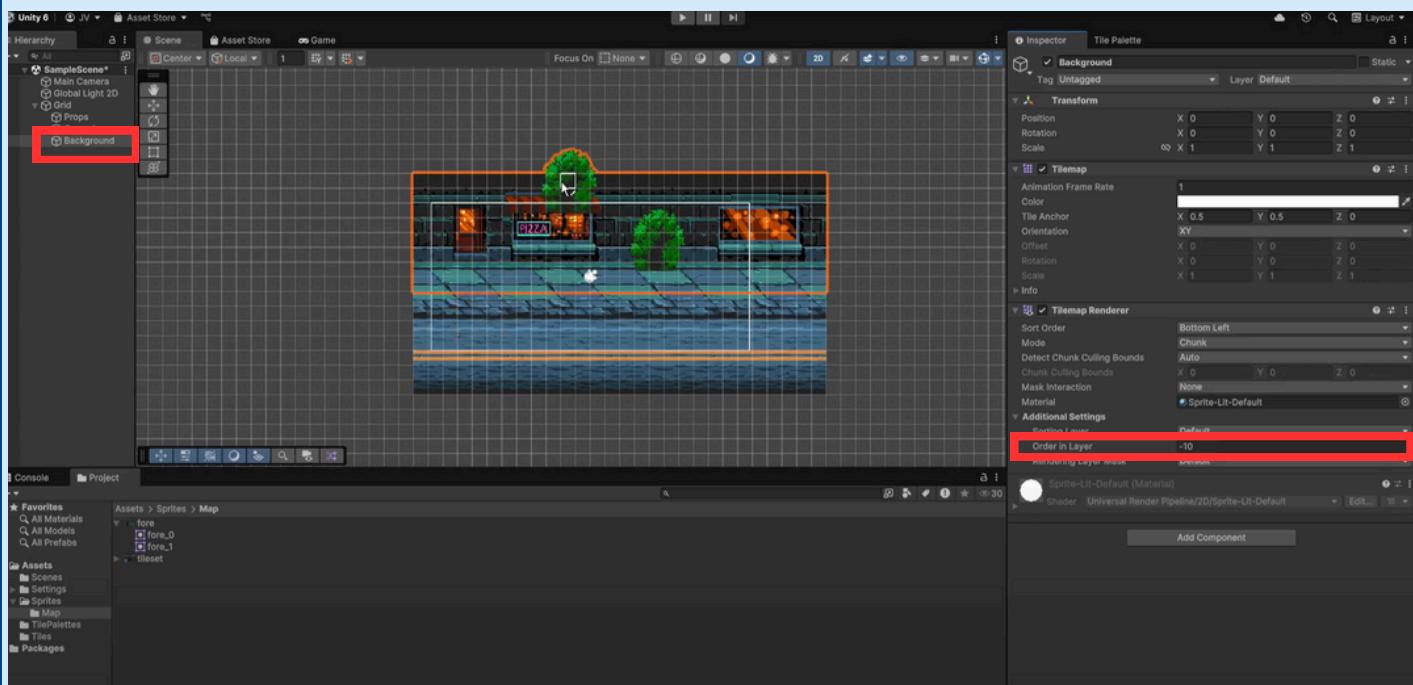
3- No final, aplicamos uma “Platform Effector 2D” ao chão. Esta componente funciona para definir sentidos em que a plataforma deixa ou não atravessar. Pensa em como funcionam as plataformas no jogo do Super Mario, é possível saltar e passar pela plataforma e não cair :)

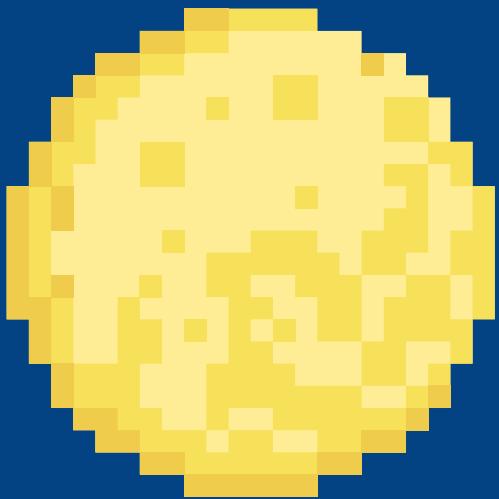


## Criação de Mapa, aplicação de ordem nas layers

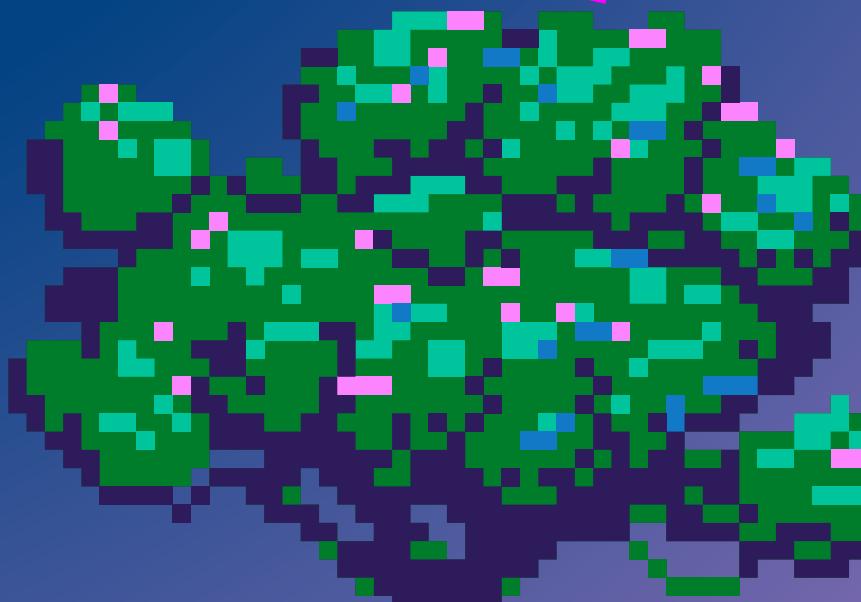
Mesmo num jogo 2D, conseguimos aplicar ordem nos objetos. Se quisermos algum objeto em frente de outro, temos de usar “Order in Layer”, funcionando como o eixo z.

Neste caso, se quisermos o background por baixo de todos os objetos, aplicamos uma Order in Layer de -10.



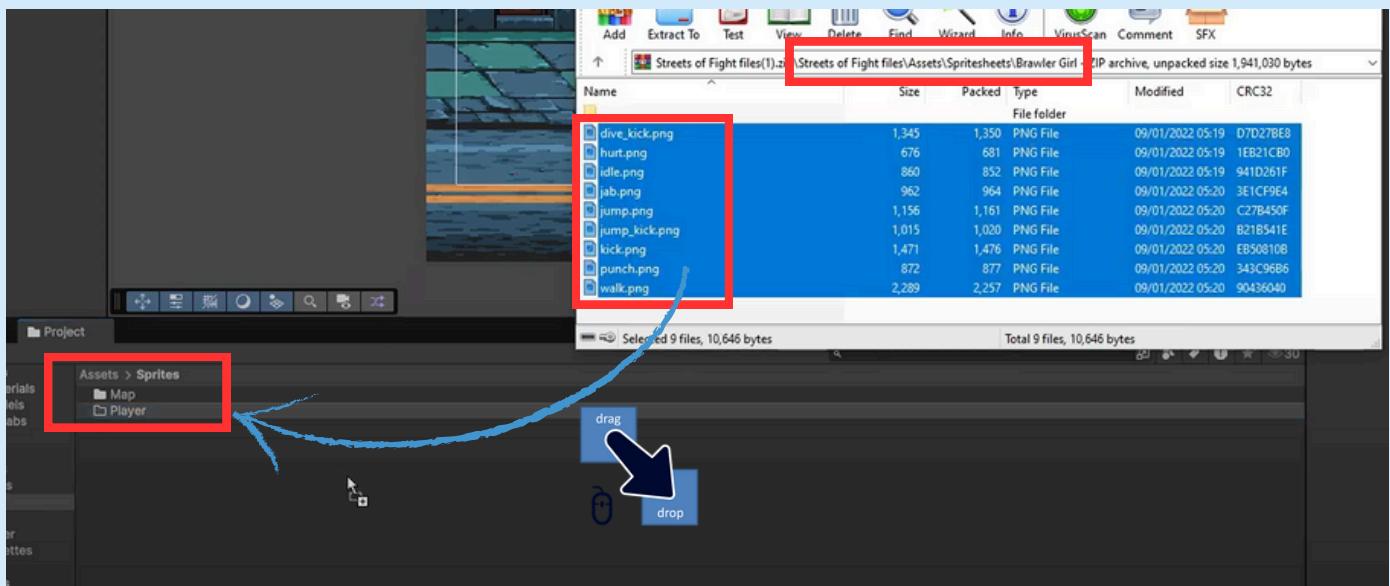


PLAYER  
MOVEMENT

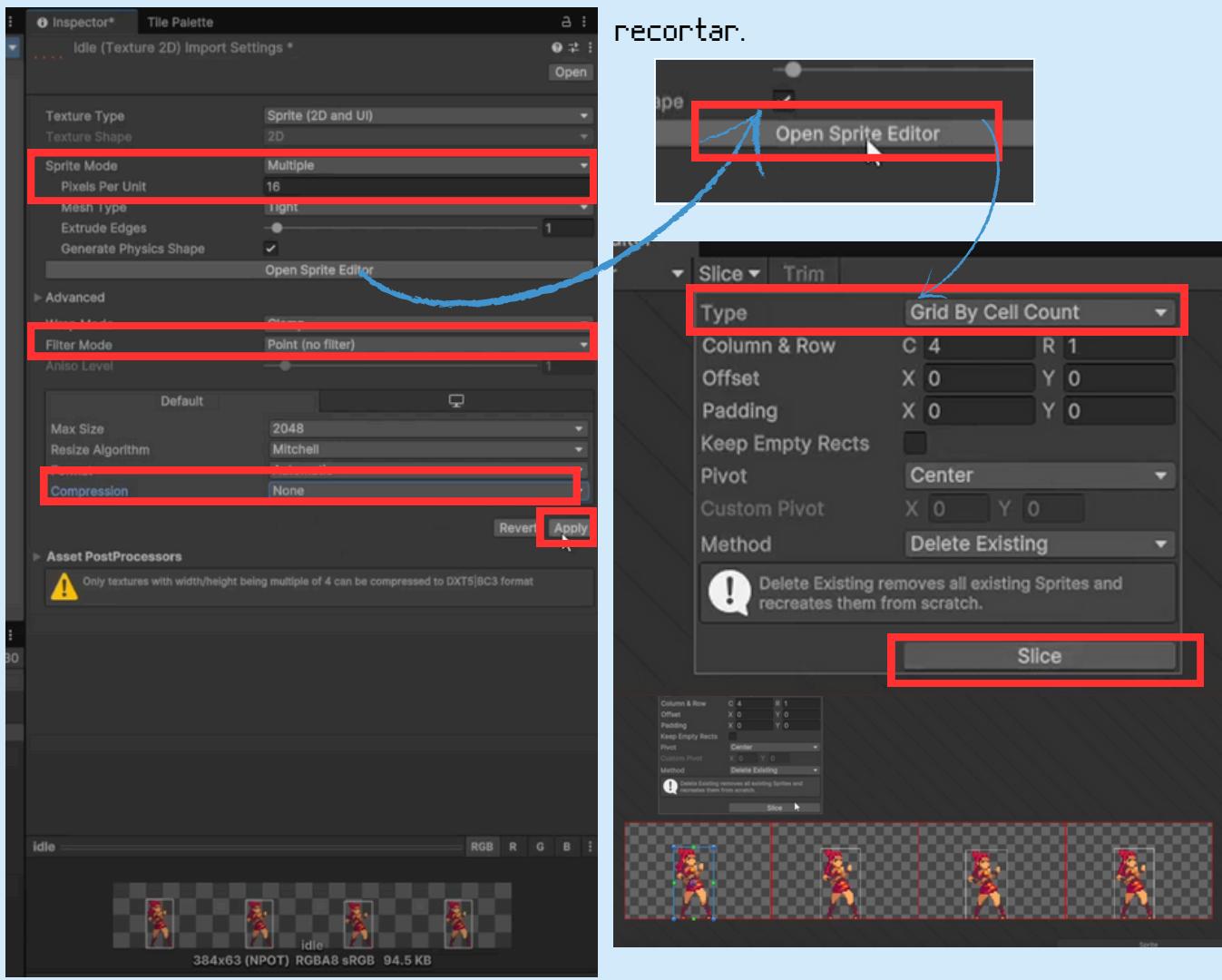


## Criação do Jogador, aplicação de Assets

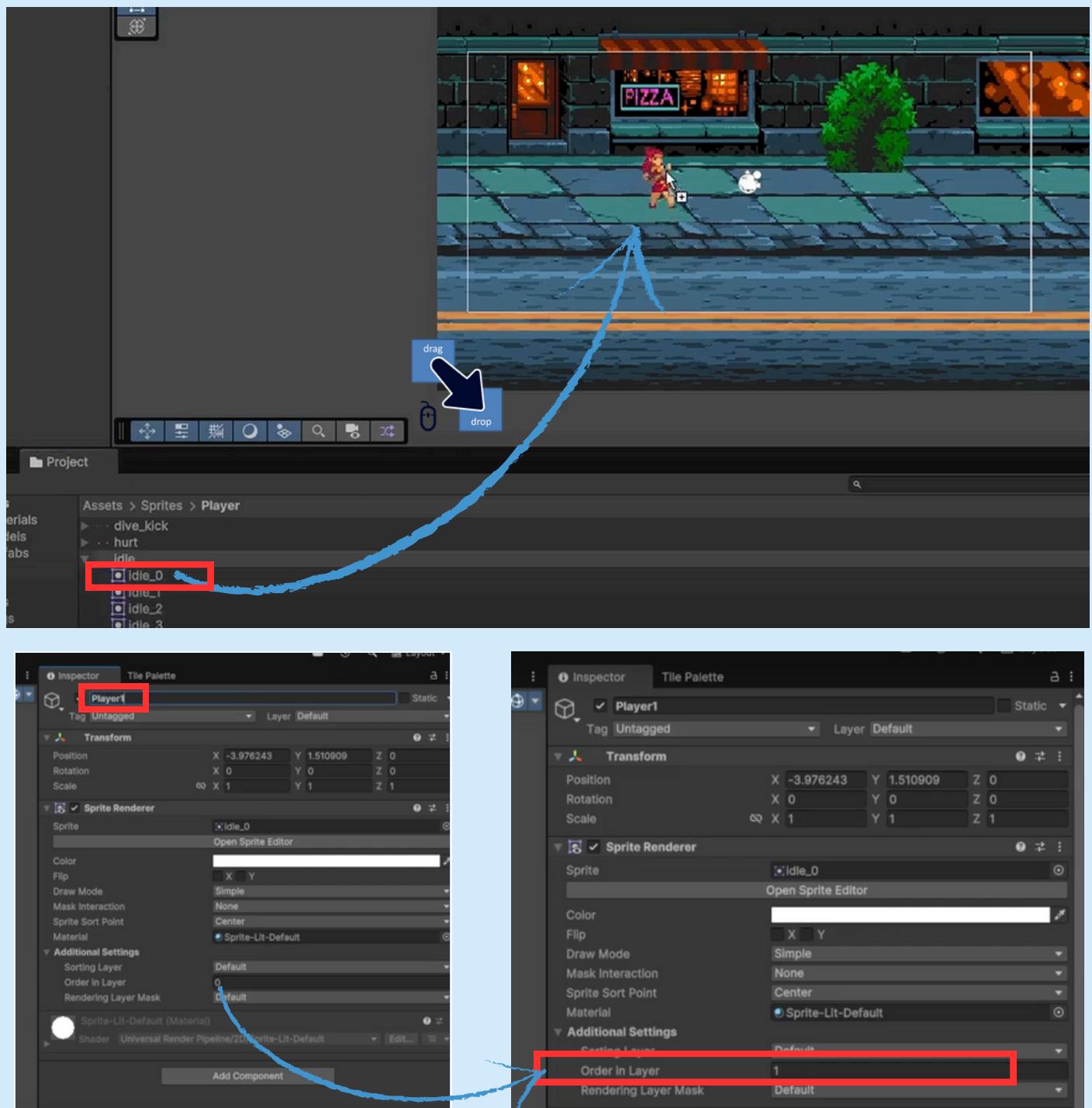
1- Para criarmos o jogador, primeiro temos de adicionar uma pasta de sprites do Player dentro da pasta de Assets. Depois disso, dentro do zip de Assets Streets of Fight, arrastamos todas as imagens de “Brawler Girl” para a pasta do Player.



2- A seguir, temos de recortar os Sprites do Player, começando pelo Idle, tal como fizemos para o Tilemap. Alterar os seguintes aspetos, aplicar mudanças e abrir o Sprite Editor para recortar.



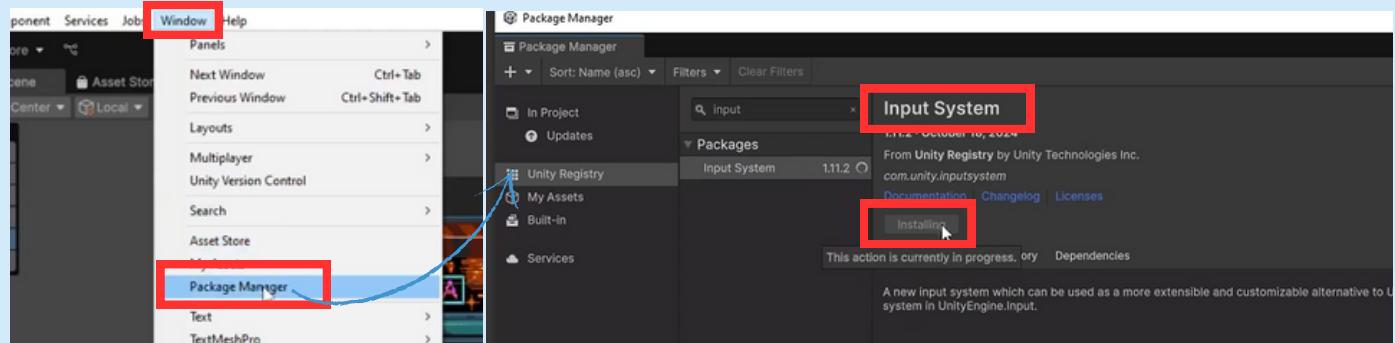
3- Arrastamos o sprite idle\_0 para dentro do cenário, alteramos o nome para Player1 e metemos uma order in layer de 1. Assim temos o nosso primeiro jogador na Scene.



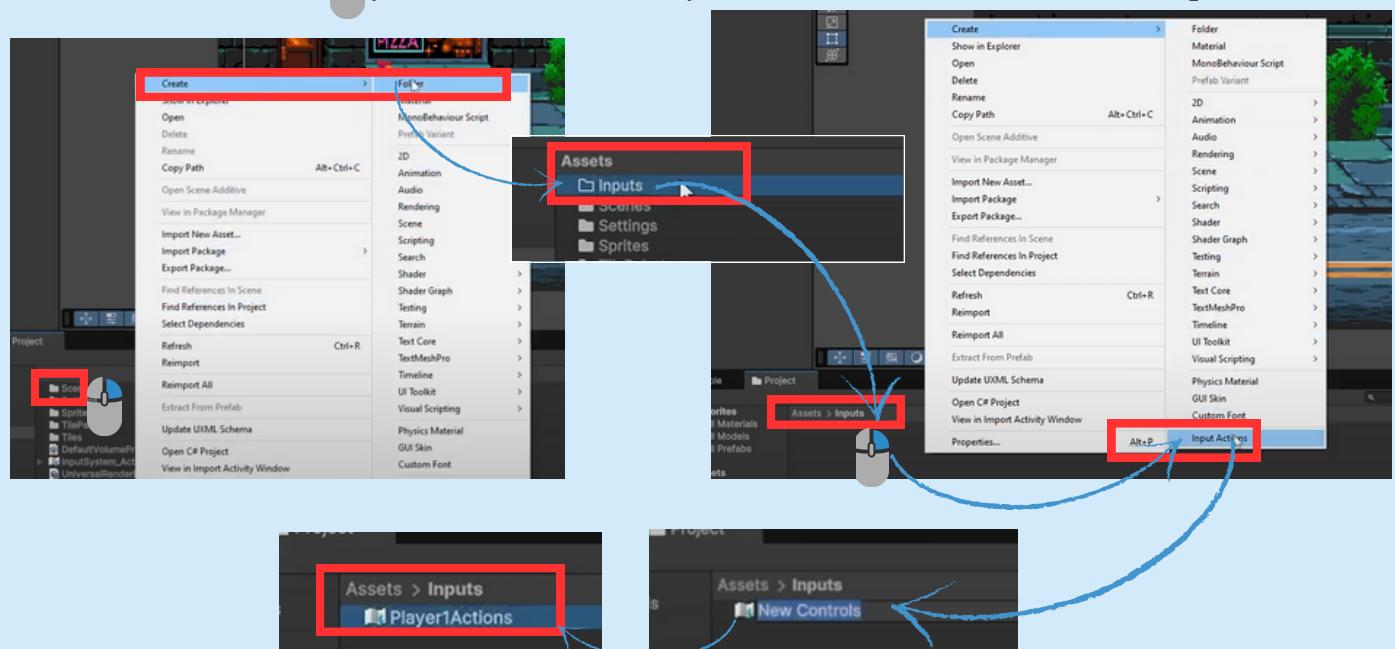
# Criação do Jogador, movimentos input e script

1- Para termos um Jogador em movimento, precisamos de associar os movimentos a um input.

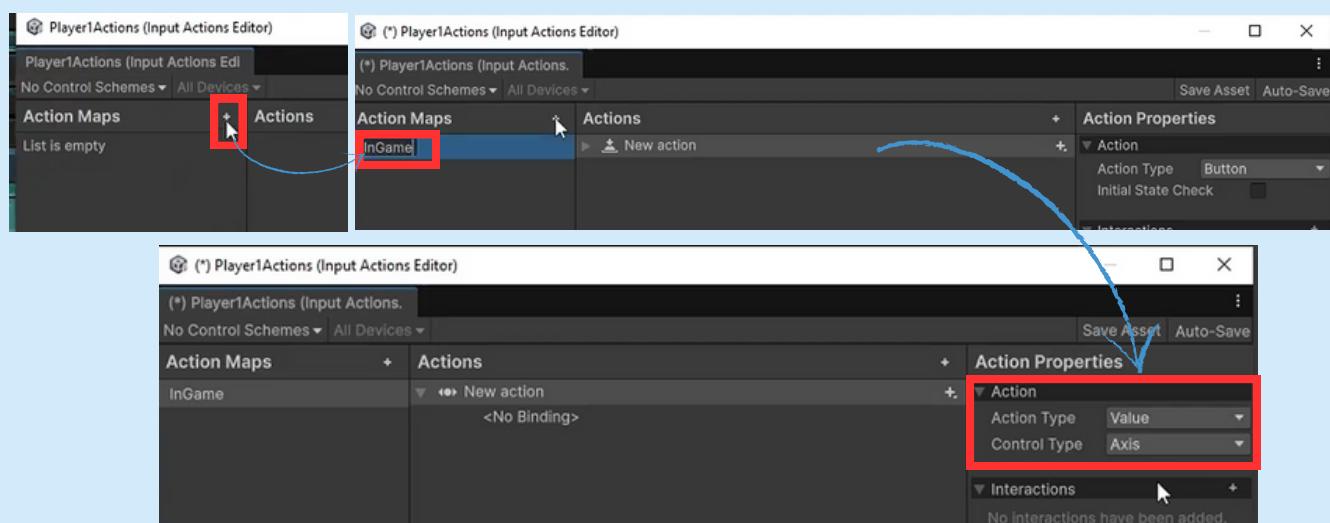
Para isso, temos de ir a “Window”->“Package Manager” e instalar a package “Input System”.



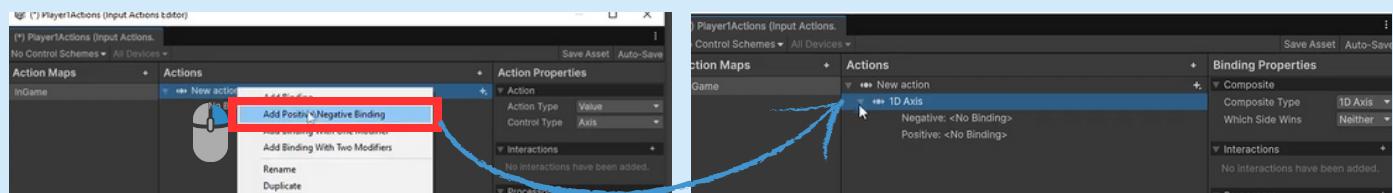
2- A seguir, criamos uma pasta dentro de Assets chamada “Inputs”, dentro dessa pasta, como botão direito do rato , criamos uma nova “Input Action” e damos o nome de “Player1Actions”.



3- Se clicarmos duas vezes na “Player1Actions”, isto irá abrir uma nova tab chamada “Input Action Editor”. Neste editor, queremos adicionar uma nova Action Map chamada “InGame” e alterar “Action Type” para Value e “Control Type” para Axis.

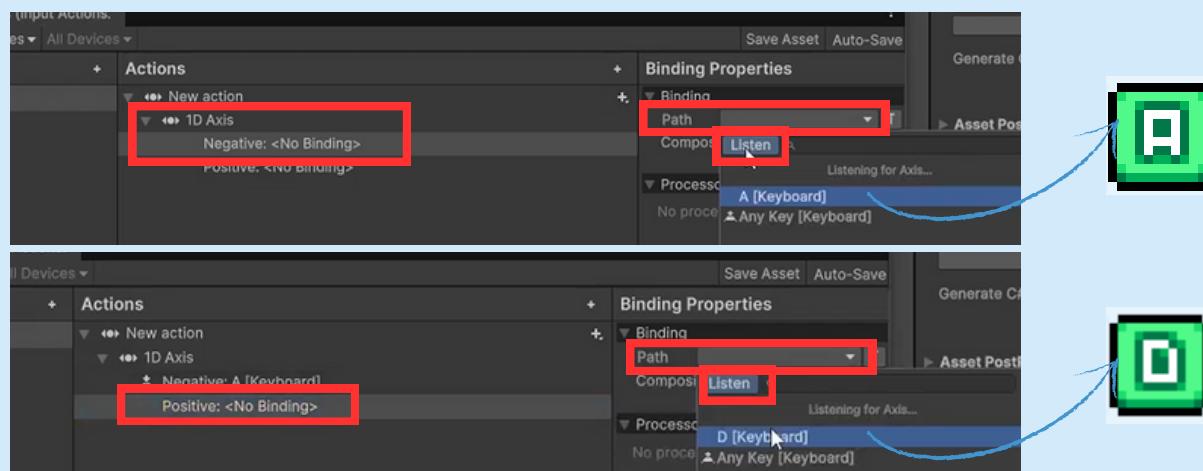


4- Após isso, clicamos com o rato direito  na action e escolhemos “Add Positive\Negative Binding”.

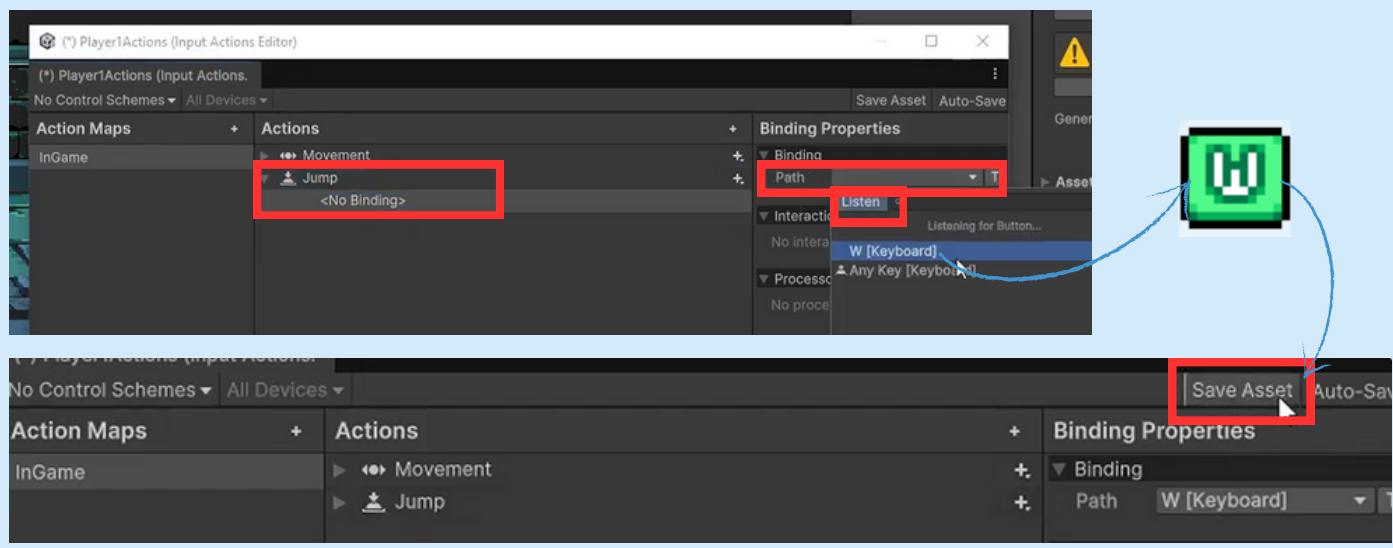
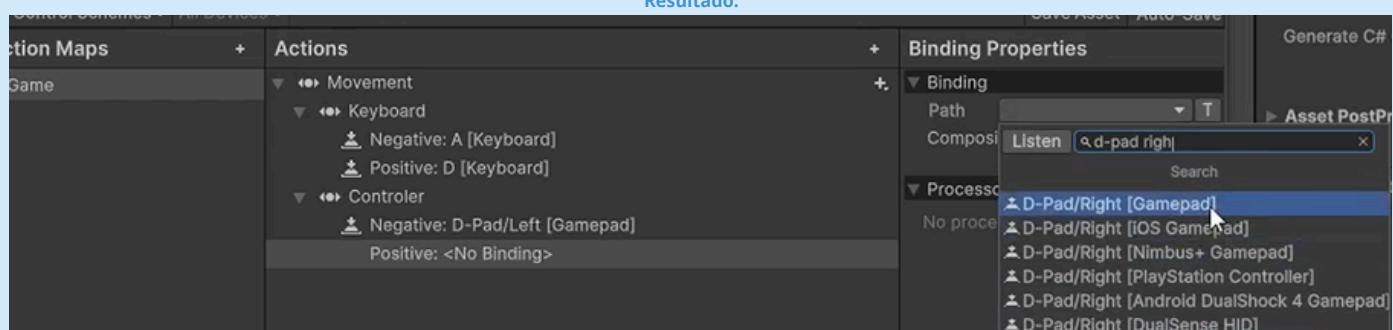


5- Para associarmos ao teclado, escolhemos o Negative, clicamos em Path e clicamos em Listen. Isto irá esperar que cliquemos numa tecla para associar, neste caso, a tecla A.

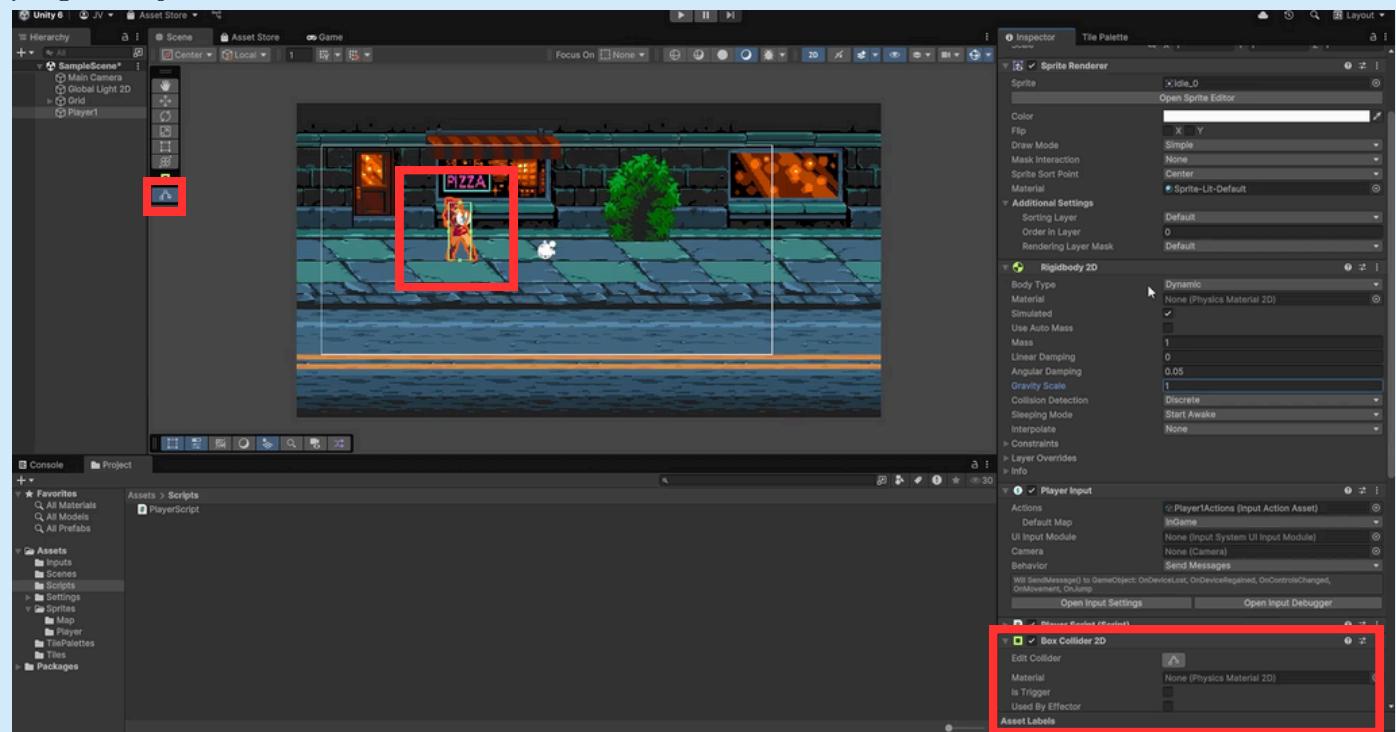
Para associar as outras teclas, fazemos o mesmo. Positive associamos à tecla D e também é possível associar ao gamepad. Após isso adicionamos outra Action chamda Jump e associamos ao W. No fim, clicar no “Save Assets” para guardar as mudanças.



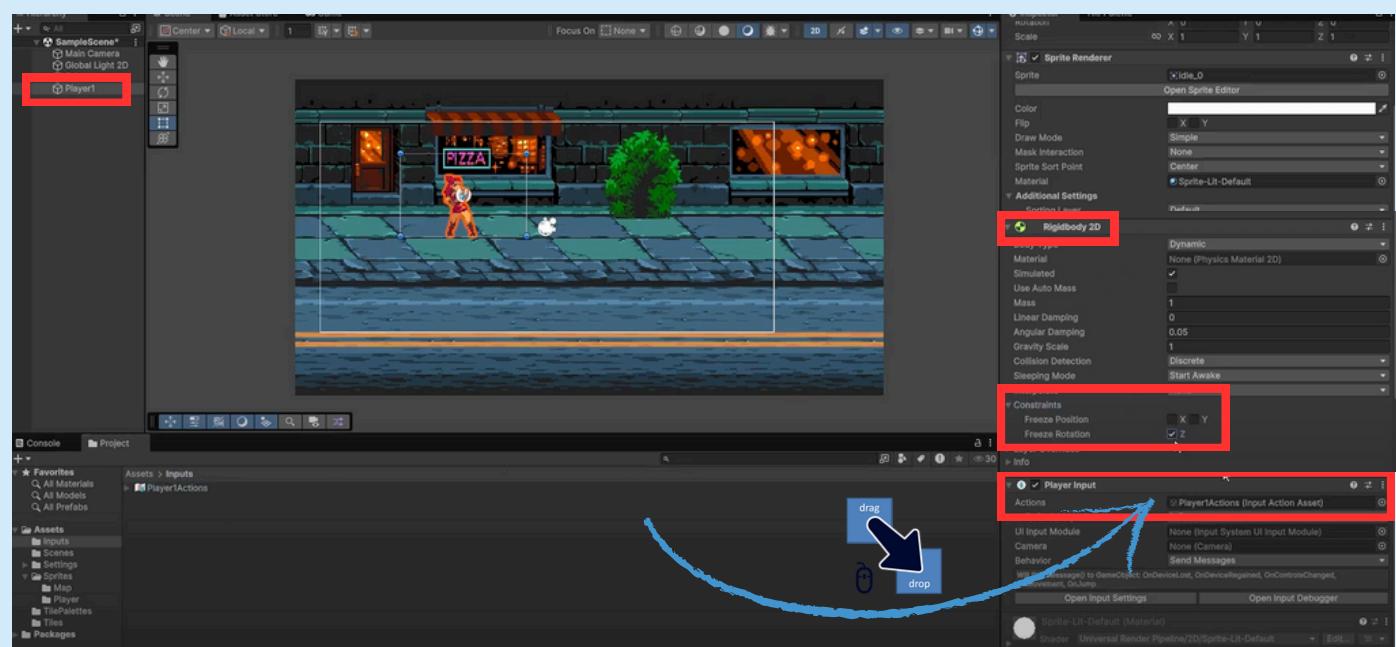
Resultado:



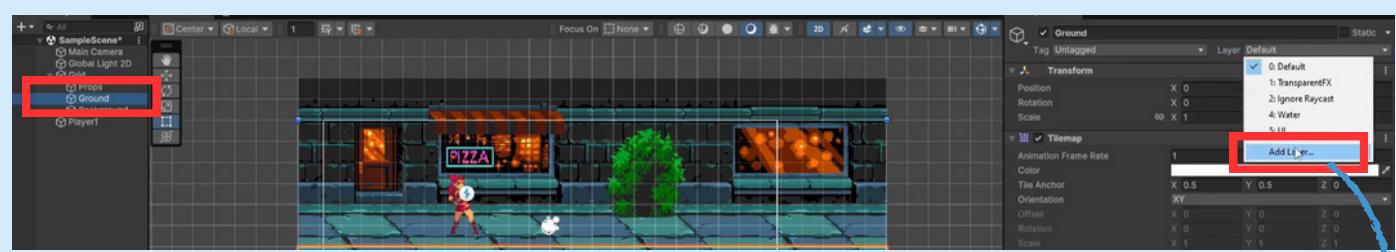
6- Para acontecer e detetar colisões, necessitamos de adicionar um “Box Collider 2D” ao player e ajustar o tamanho do collider.

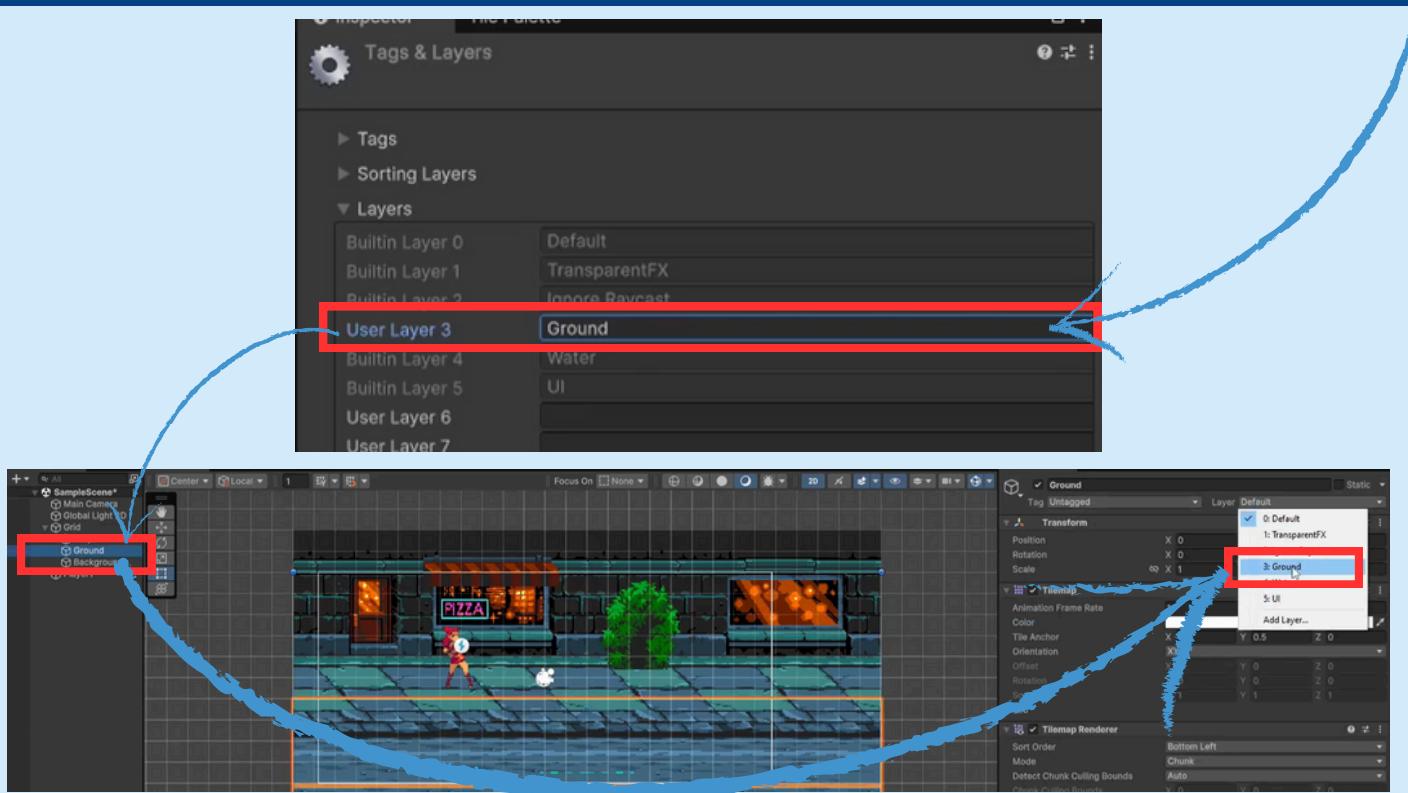


7- Selecionamos agora o Player1, adicionamos uma componente de “Rigidbody 2D” com freeze rotation no eixo do z, e um “Player input”. Após isso, arrastamos a nossa Player1Actions para onde temos a “Action”.

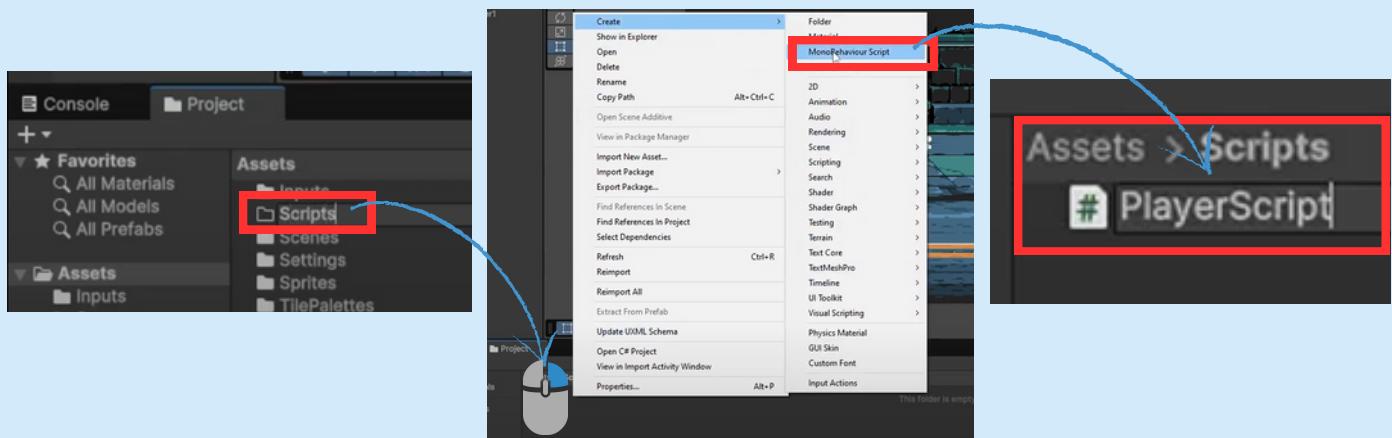


8-No Unity, é importante termos layers. Estes layers são importantes para definirmos que objects (física, colliders, rigidbody) podem ou não interagir com os outros. Para isso, adicionamos uma layer chamada “Ground” e depois associamos esta layer ao object Ground.

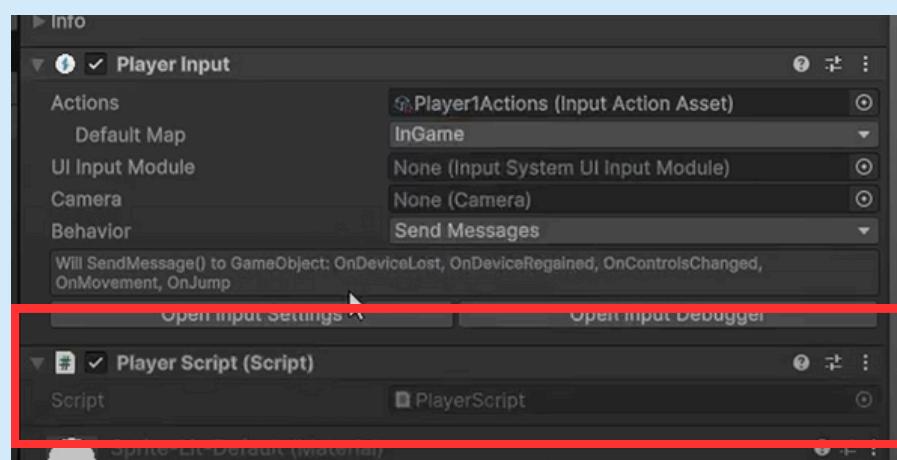




8- Para dar movimento ao jogador, precisamos agora de adicionar um script. Começamos por criamos uma pasta de “Scripts” dentro de Assets e criamos uma “MonoBehaviour script” chamada “PlayerScript”. Depois, clicamos duas vezes nela para abrir uma janela onde vamos escrever o nosso código (explicação do código passo a passo na playlist).



9- No fim, adicionamos ao player a componente de PlayerScript e o player já consegue mover.



## PlayerScript v1:

```
using UnityEngine;
using UnityEngine.InputSystem;
using System.Collections;

public class PlayerScript : MonoBehaviour
{
    private float movement;
    [SerializeField] private float speed = 5;
    [SerializeField] private float jumpForce = 10;

    private Rigidbody2D rb;
    private Collider2D coll;
    private LayerMask groundLayer;

    private void Awake() {
        rb = GetComponent<Rigidbody2D>();
        coll = GetComponent<Collider2D>();
        groundLayer = LayerMask.GetMask("Ground");
    }

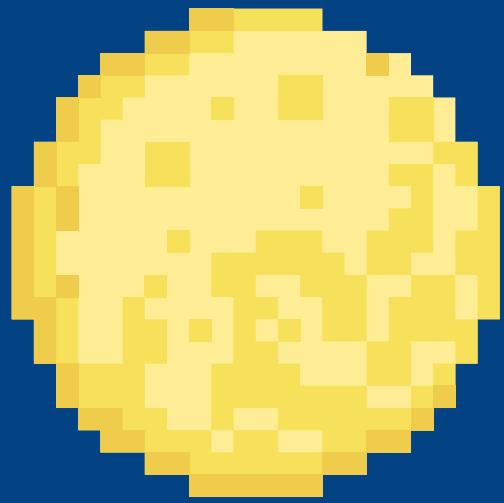
    private void OnMovement(InputValue value) {
        movement = value.Get<float>();
    }

    private void OnJump(InputValue value) {
        if (IsGrounded()) {
            rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
        }
    }

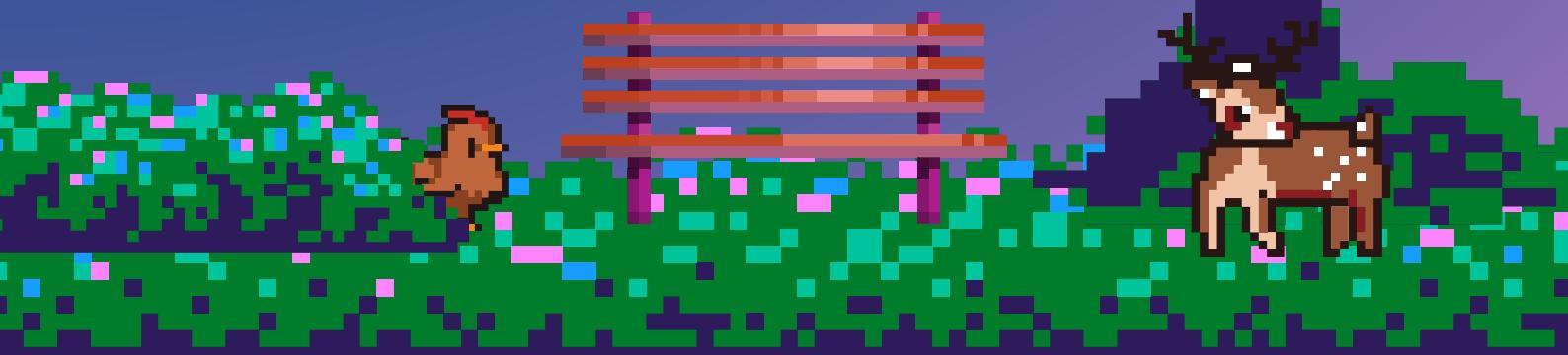
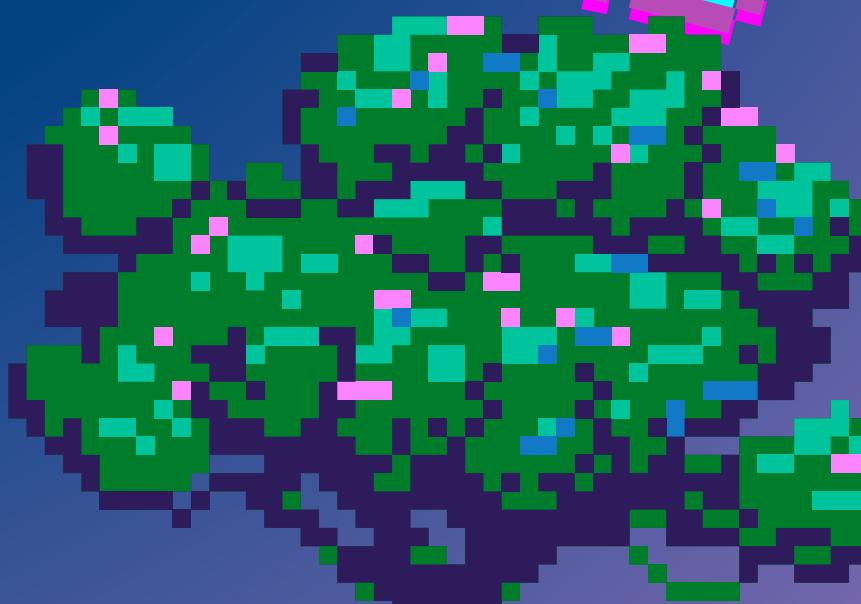
    private void FixedUpdate() {
        rb.linearVelocity = new Vector2(movement * speed, rb.linearVelocity.y);
        UpdateFacingDirection();
    }

    private void UpdateFacingDirection() {
        if (movement != 0 && transform.localScale.x != movement)
            transform.localScale = new Vector3(transform.localScale.x * -1, transform.localScale.y, 0);
    }

    private bool IsGrounded() {
        return Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0, Vector2.down, 0.1f, groundLayer);
    }
}
```

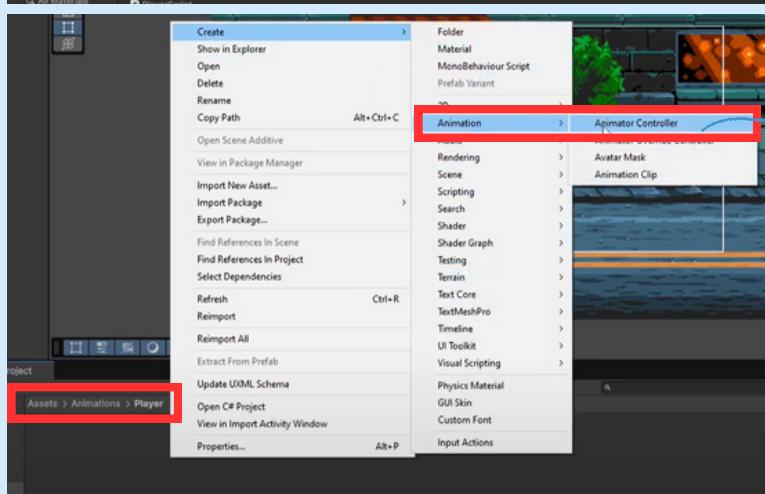
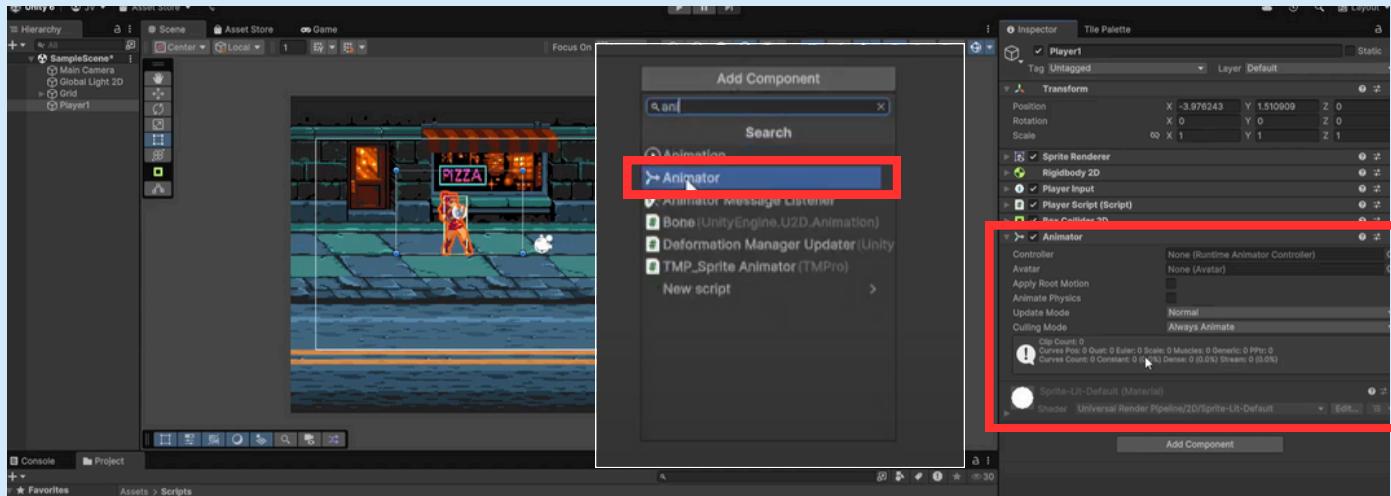


# ANIMATIONS

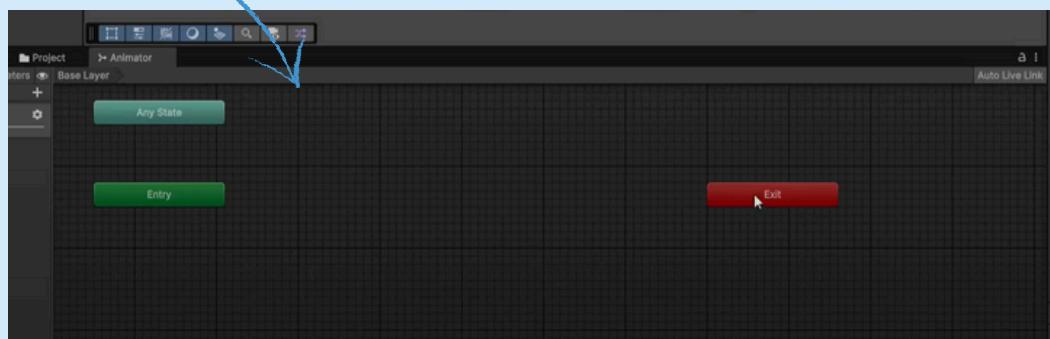
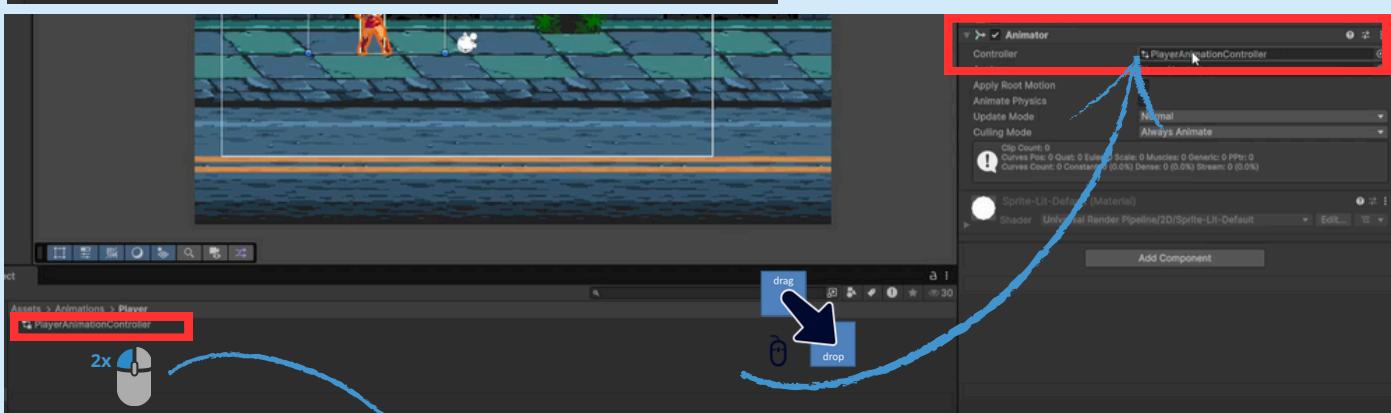


## Criação do Jogador, animações

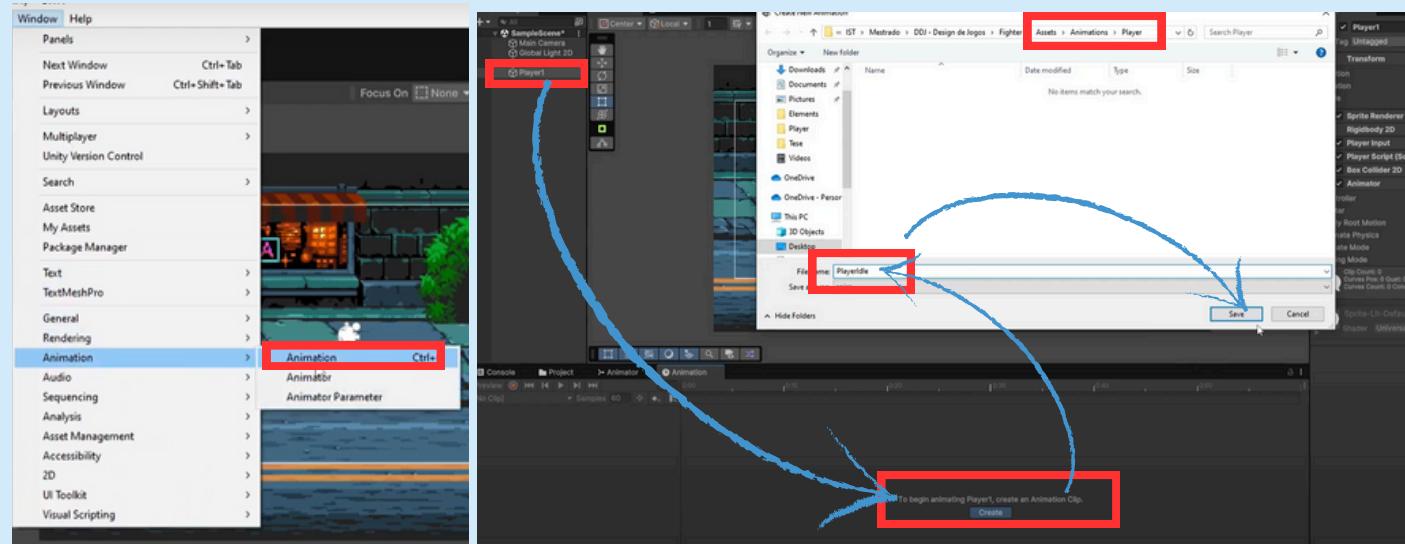
1- Para animarmos os movimentos do jogador, primeiro adicionamos ao player uma nova componente chamada “Animator”. Depois, criamos um “PlayerAnimationController” (dentro de Assets>Animations>Player) e arrastamo-lo para dentro do Controller de Animator.



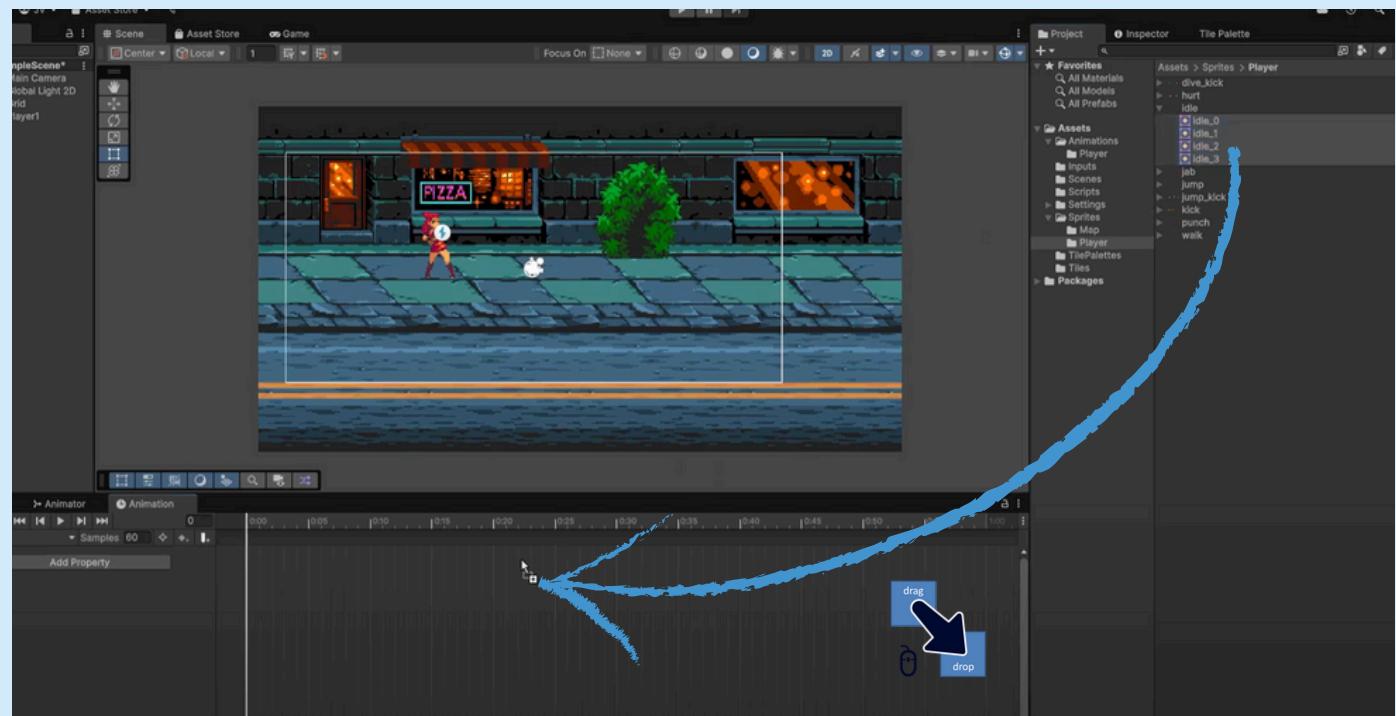
Assets > Animations > Player  
PlayerAnimationController



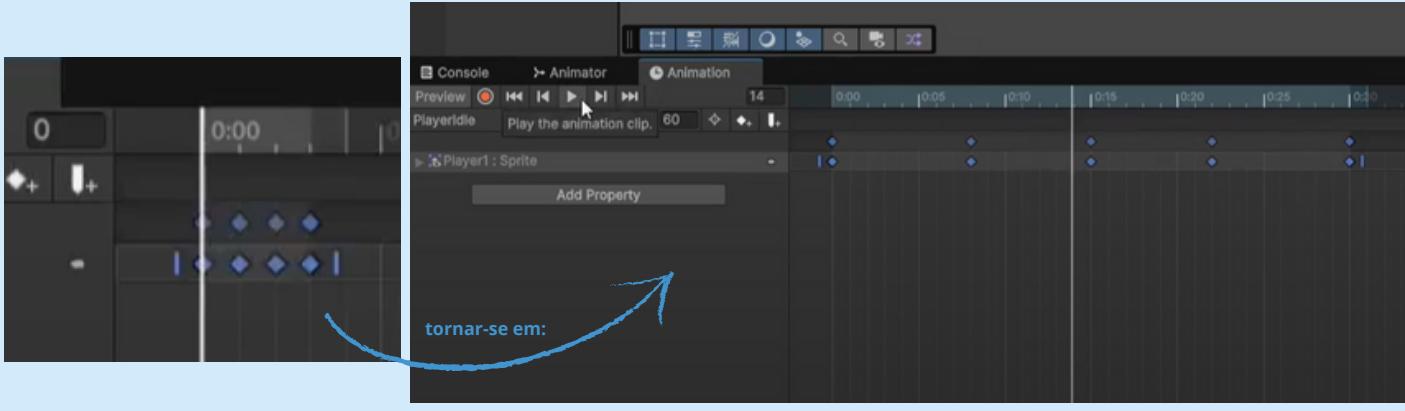
2- Para criarmos as animações, abrimos a tab de Animation, selecionamos o Player 1, e carregamos no botão Create e criamos a animação do “PlayerIdle”.



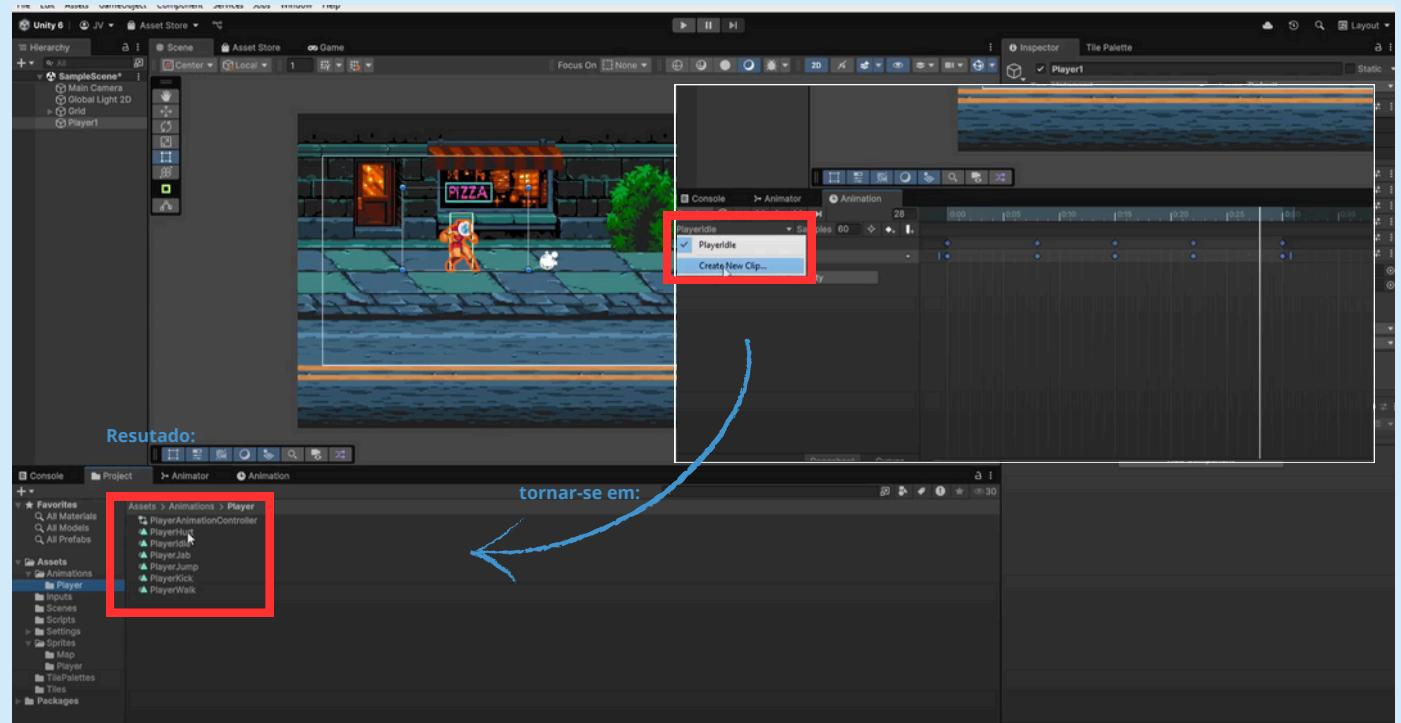
3- A seguir, arrastamos todos os sprites de idle para dentro da Animation PlayerIdle e ajustamos a animação em si (repetir o primeiro segmento no fim e aumentar o tempo da animação).



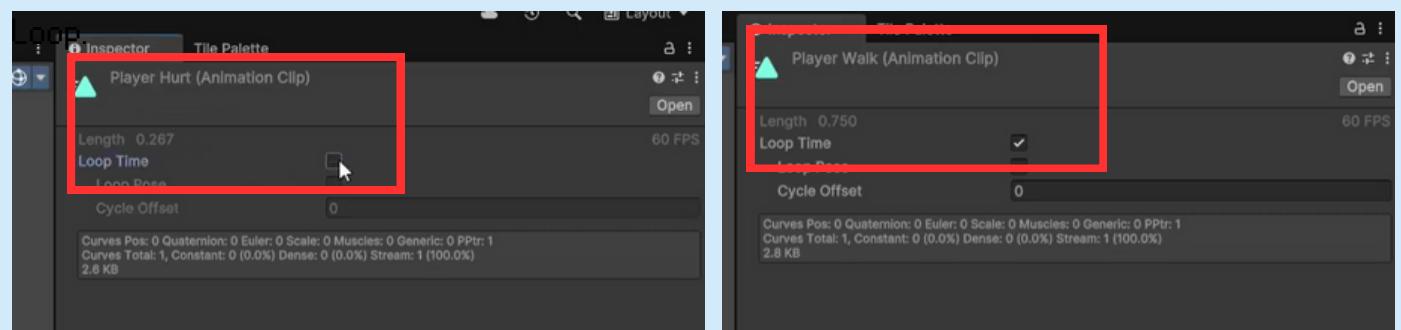
Resultado:



4- Estes passos número 2 e 3 de criação de animação deve ser repetido para todos movimentos(hurt, jab-repetir o último segmento, jump, jump\_kick, kick, punch, walk). Para recortar os sprites, visite a secção de “Criação do Jogador, aplicação de Assets”. No caso de dúvida, visite o vídeo da playlist.

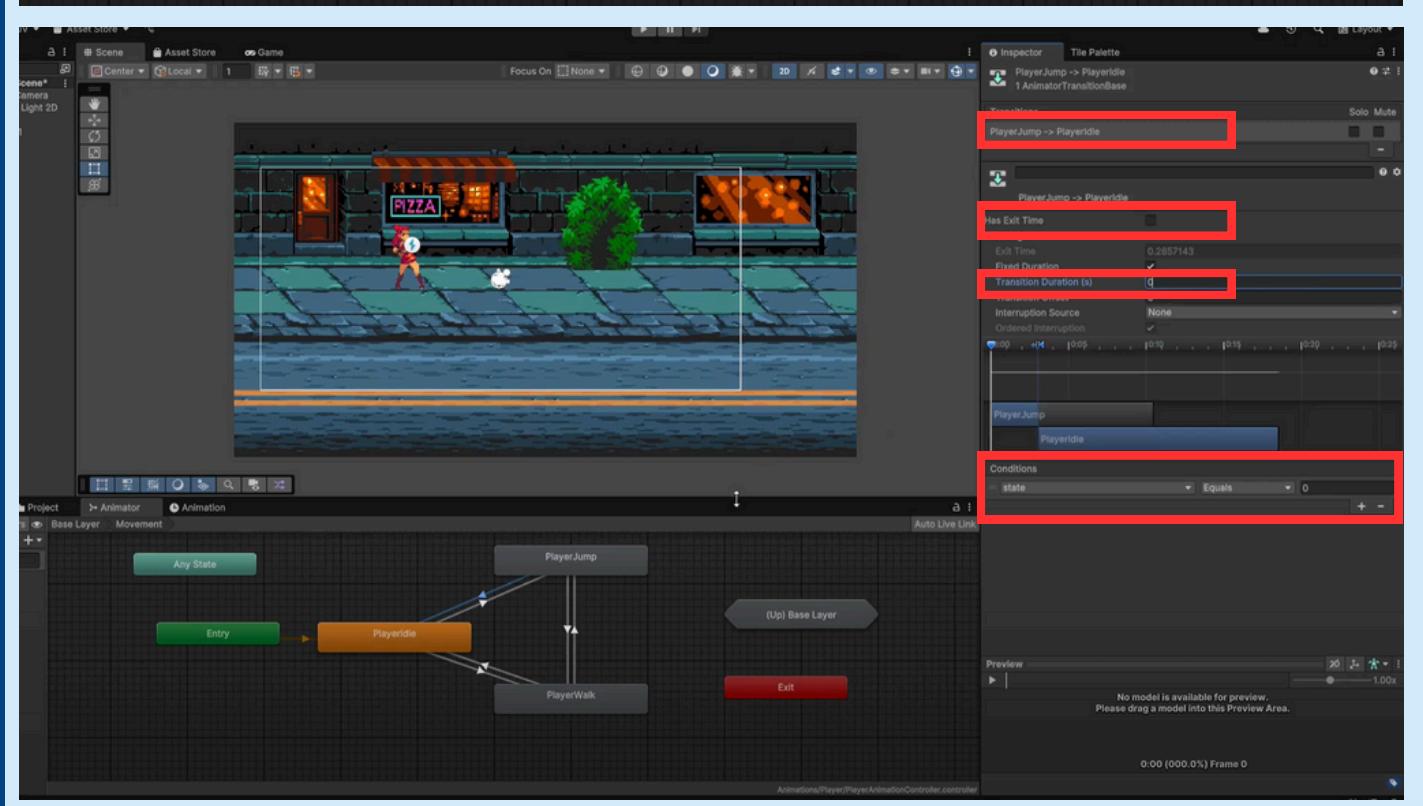
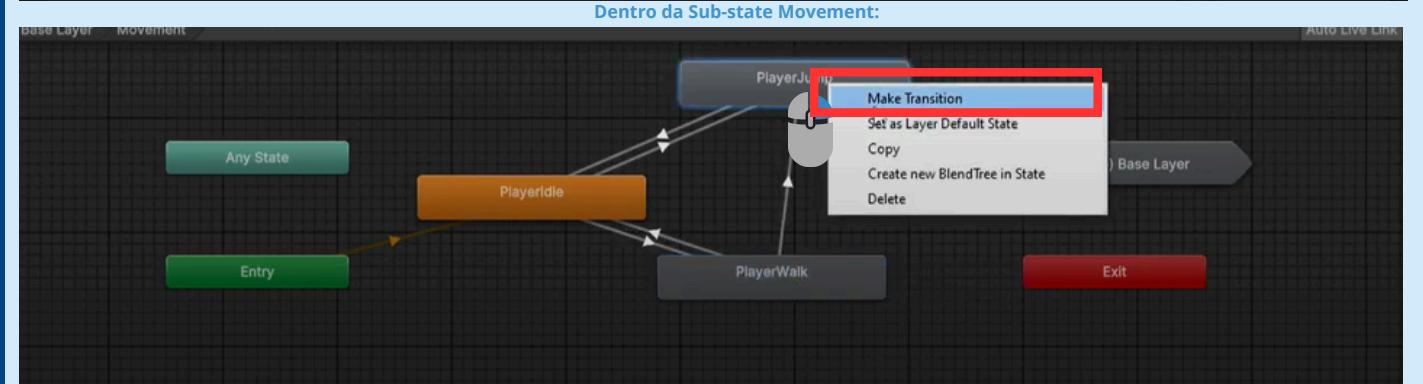
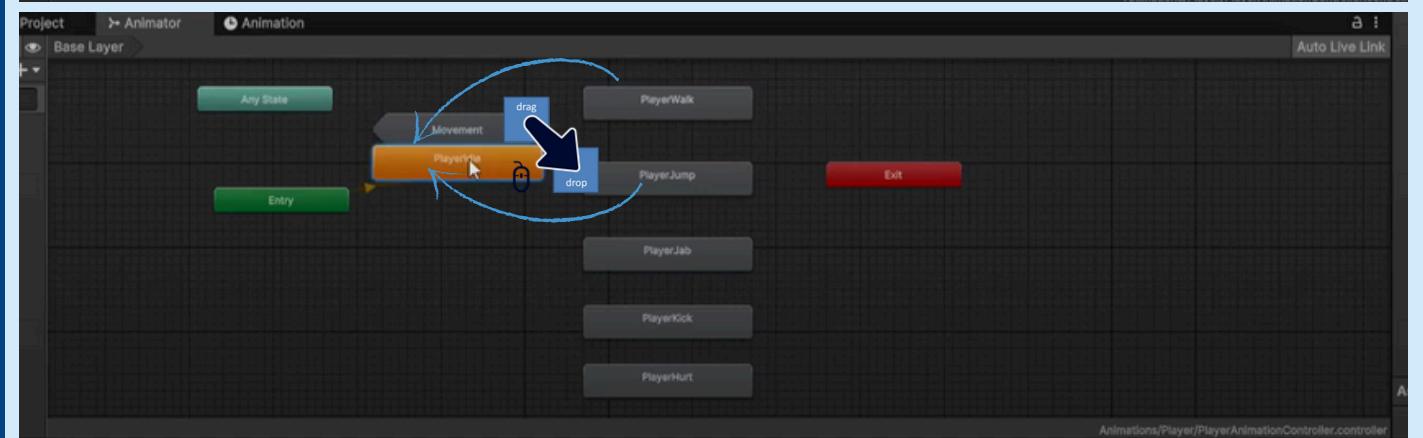
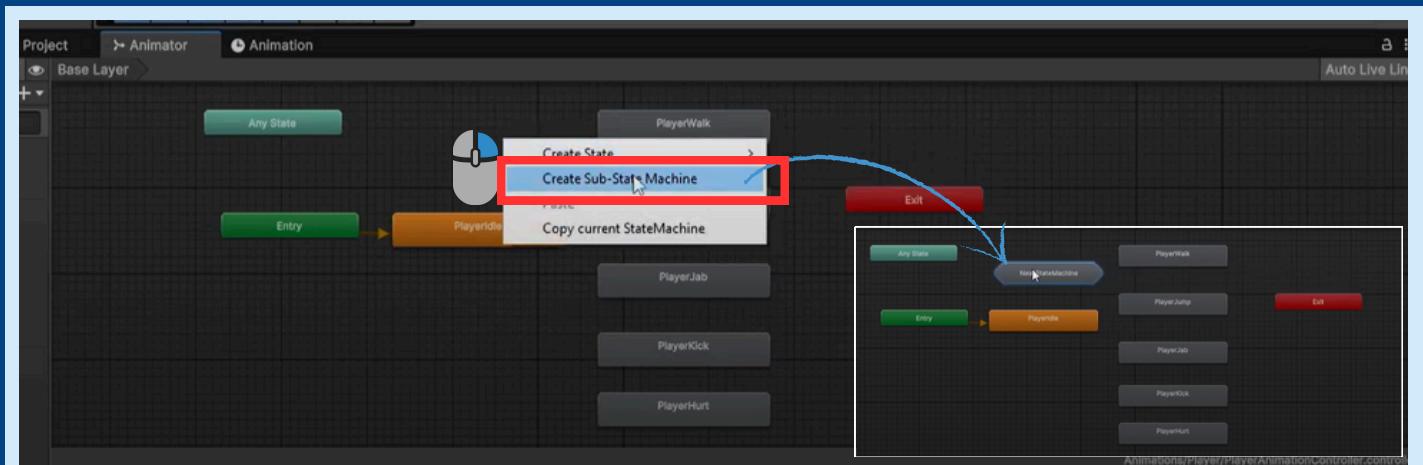


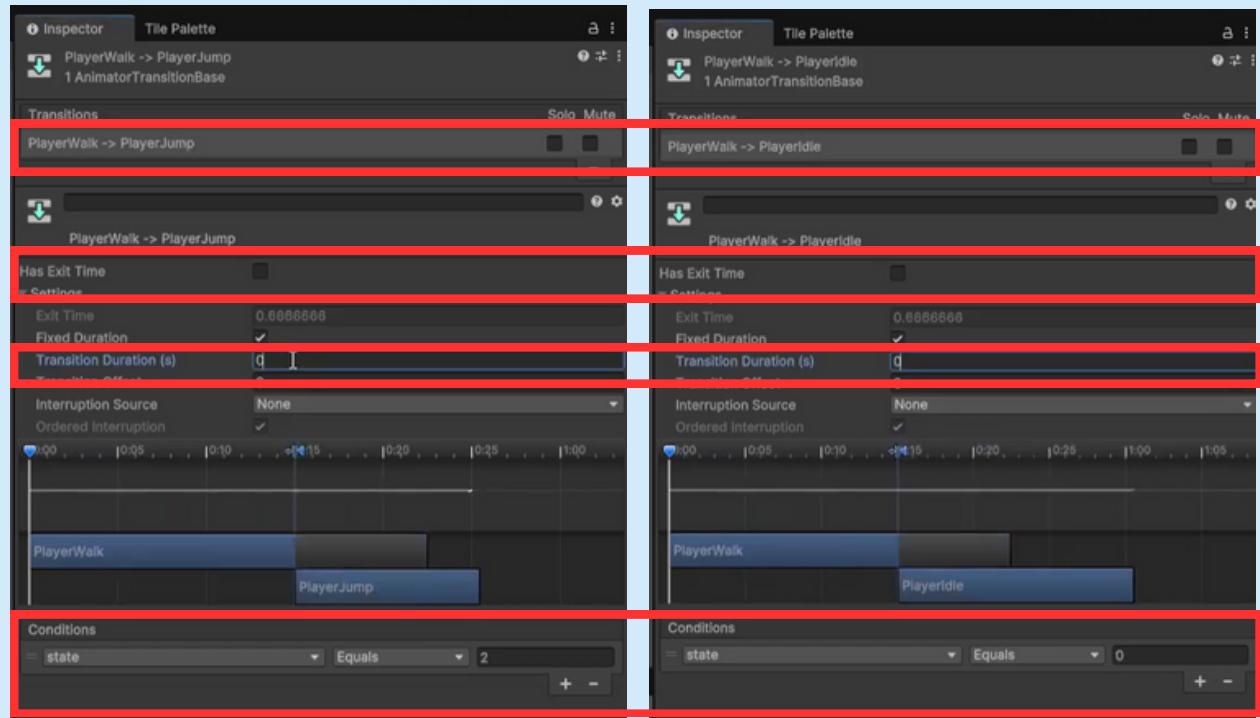
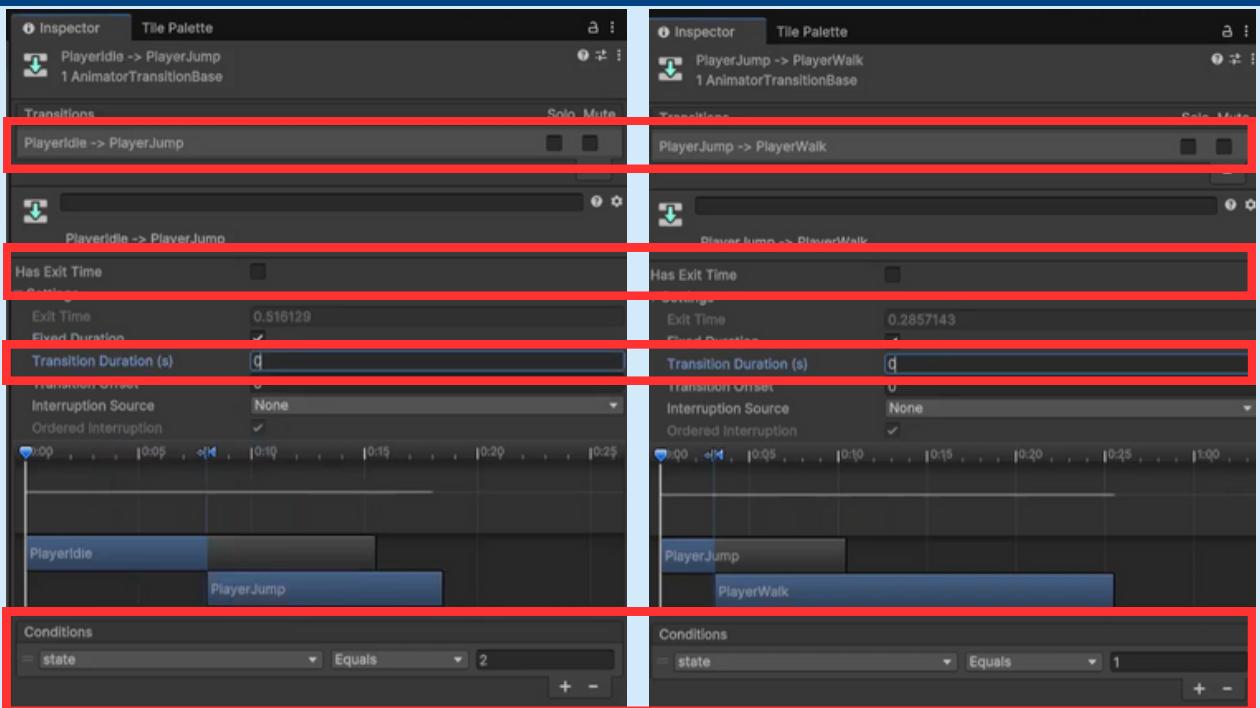
5- Por default, as animações dão loop. No entanto, queremos só as animações de Idle e Walk a dar loop. Por isso, dentro do Inspector de cada animação devemos verificar a checkbox de



6- Agora, para organizarmos as animações, voltamos ao Animator e adicionamos um int de 0 chamado state. Adicionamos uma “Sub-state Machine” chamada Movement, onde temos Idle, Jump e Walk. No fim, fazemos todas as ligações entre os movimentos de acordo com as imagens abaixo.







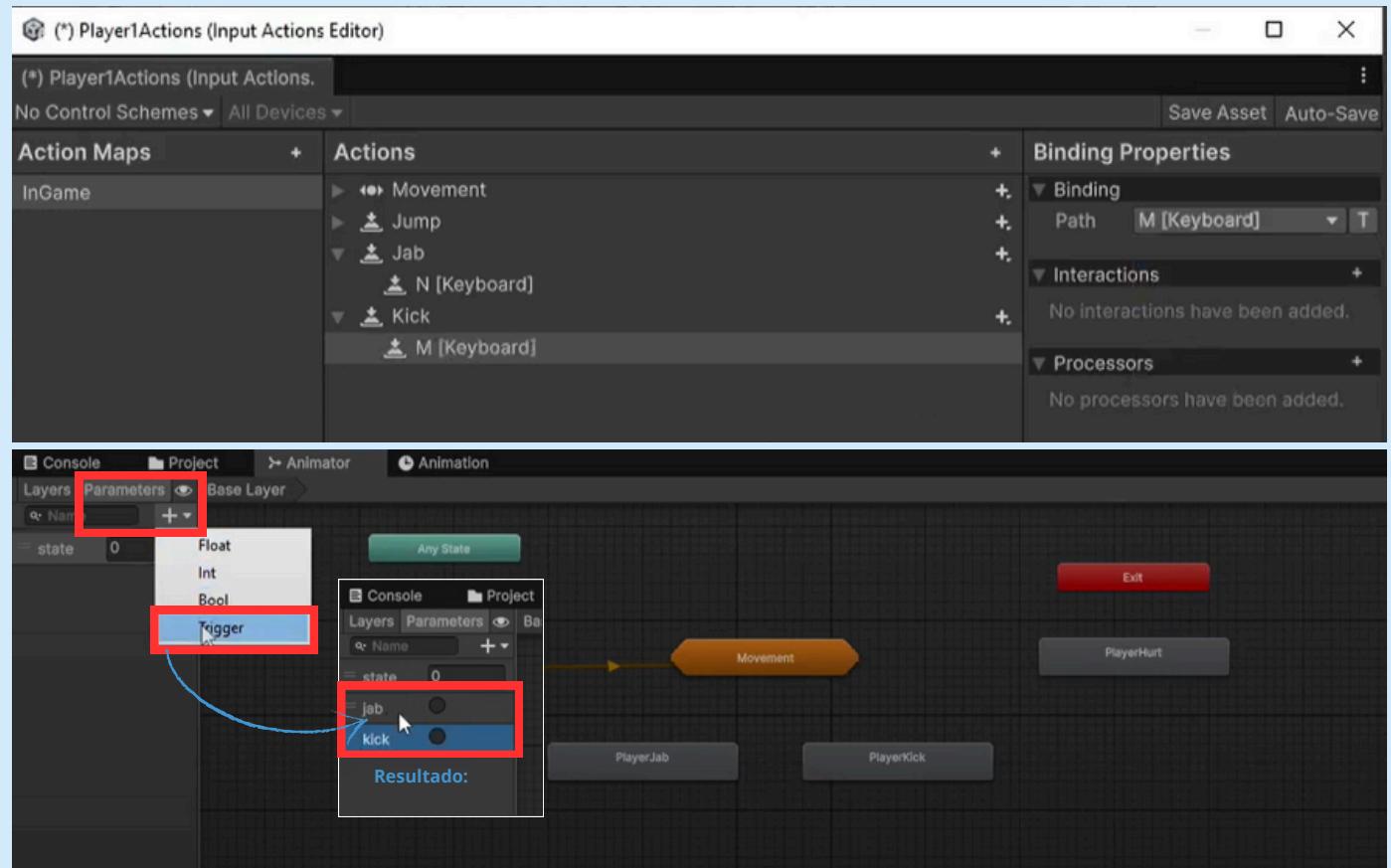
Idle, Walking, Jumping.

Segundo esta ordem, Idle é state 0, Walking state 1 e Jumping state 2.

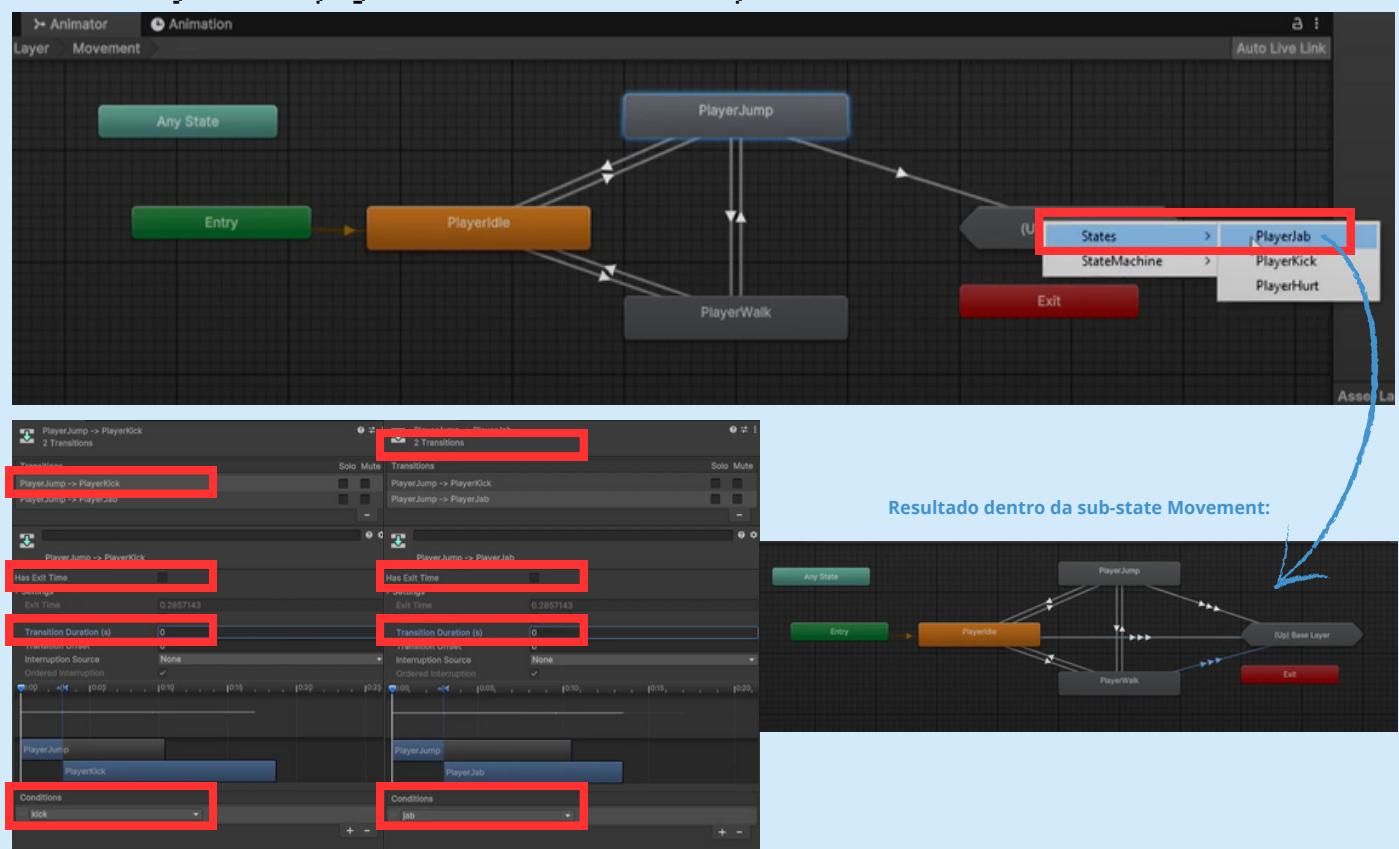
Dentro desta transição de Idle para Walk, temos uma Condition de state Equals 1. Isto quer dizer que a transição ocorre quando o state passa para 1.

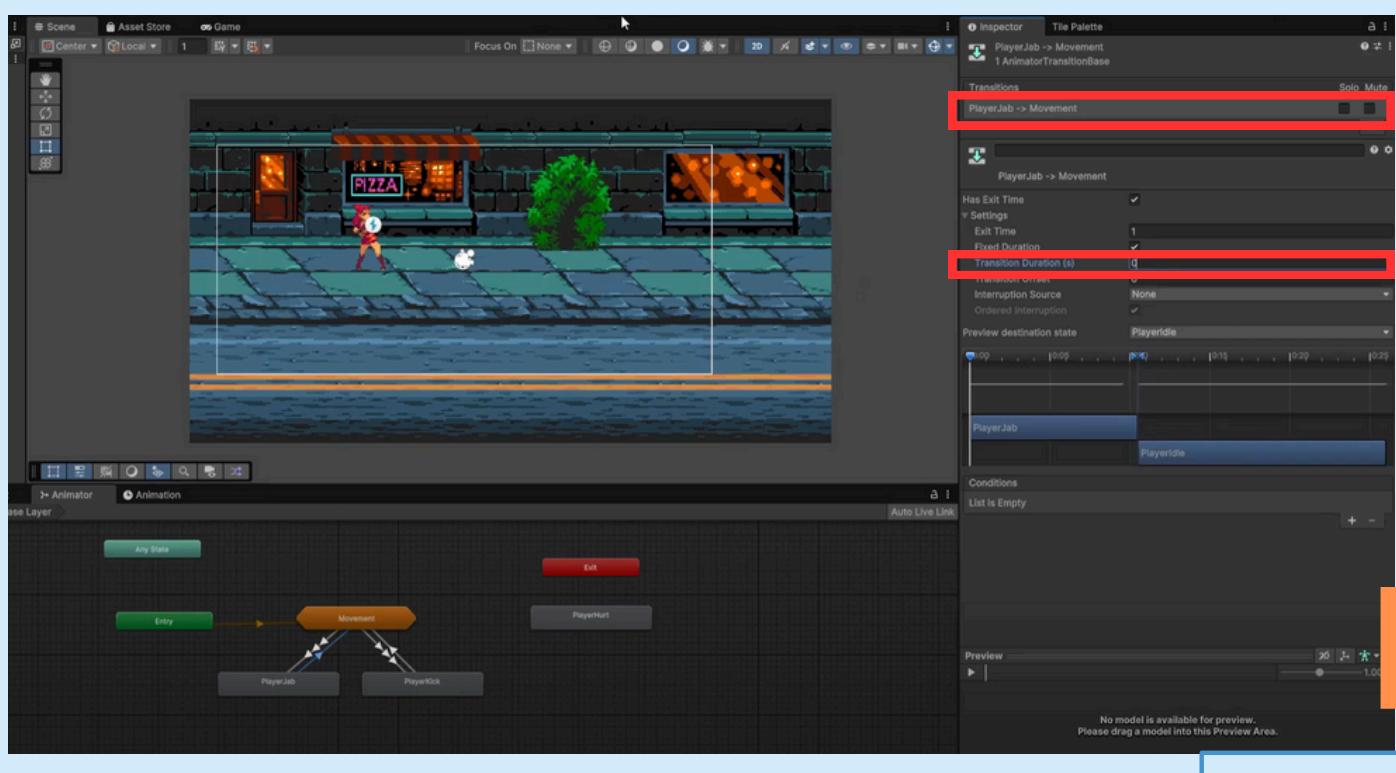
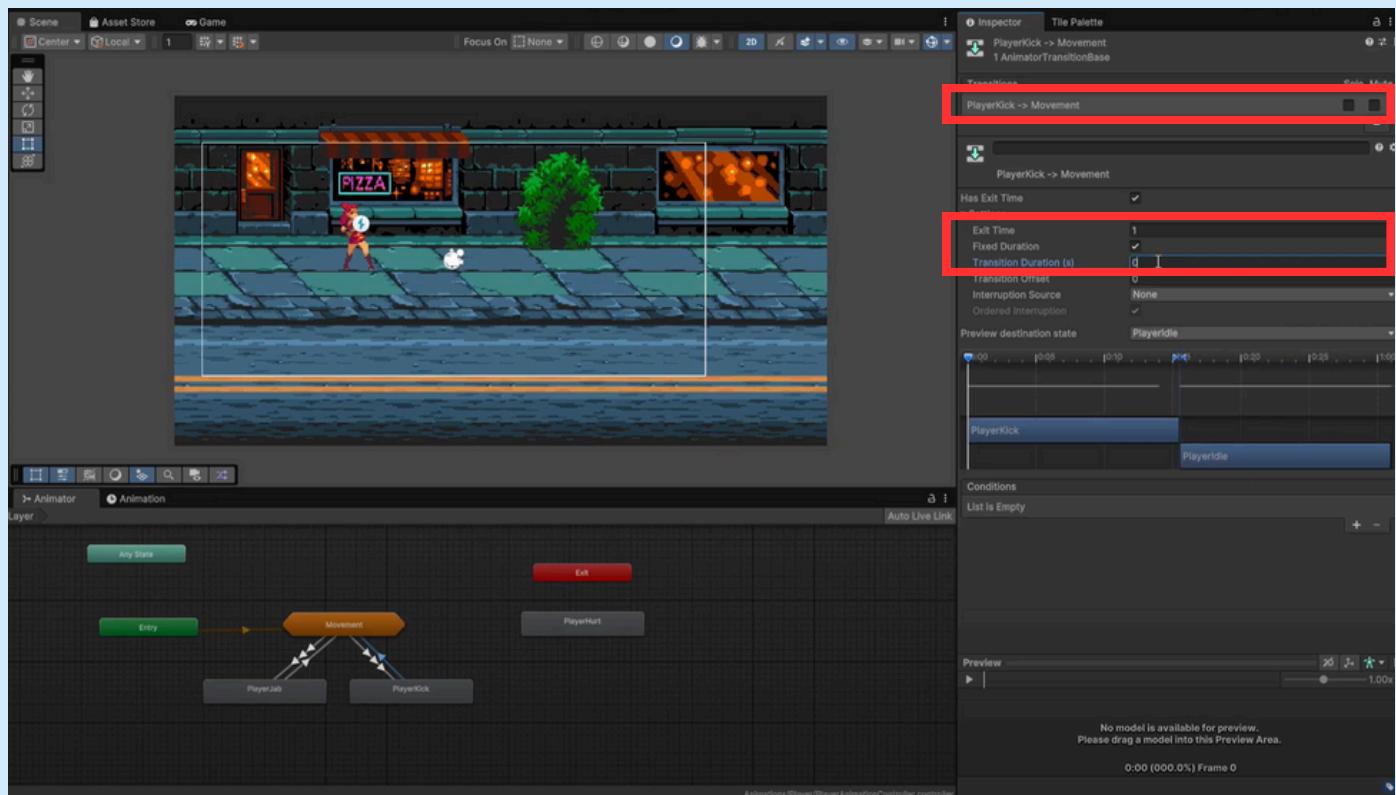
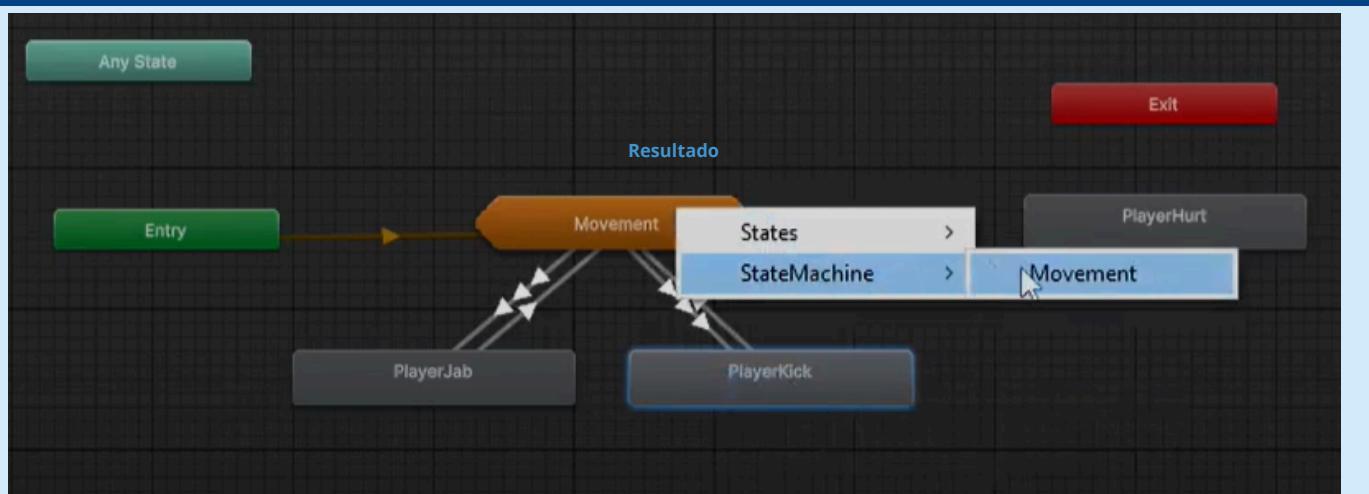
Não queremos Exit Time nem Transition time pois queremos que seja instantâneo.

7- Para animações de Jab e Kick, temos de definir primeiro os inputs destas ações, Criação do jogador, movimentos input e script. Após isso, adicionamos um parâmetro do tipo de Trigger para Jab e outro para Kick pois estes são tratados como eventos e não como states.



8- Repetimos os passos anteriores para definirmos as transições. Ligamos Jump, Idle, Walk para PlayerJab e também para PlayerKick, e na secção de Conditions, não usamos state, mas sim kick ou jab. No fim, ligamos ambas ao Movement, alterando Exit Time e Transition Duration.





## PlayerScript v2 with changes for animations:

```
using UnityEngine;
using UnityEngine.InputSystem;
using System.Collections;

public class PlayerScript : MonoBehaviour
{
    private float movement;
    [SerializeField] private float speed = 5;
    [SerializeField] private float jumpForce = 10;
    private enum AnimationState { Idle, Walking, Jumping, Falling, Kicking, Punching};

    private Rigidbody2D rb;
    private Collider2D coll;
    private Animator anim;
    private LayerMask groundLayer;

    private void Awake() {
        rb = GetComponent<Rigidbody2D>();
        coll = GetComponent<Collider2D>();
        anim = GetComponent<Animator>();
        groundLayer = LayerMask.GetMask("Ground");
    }

    private void OnMovement(InputValue value) {
        movement = value.Get<float>();
    }

    private void OnJump(InputValue value) {
        if (IsGrounded()) {
            rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
        }
    }

    private void OnJab(InputValue value) {
        anim.SetTrigger("jab");
    }

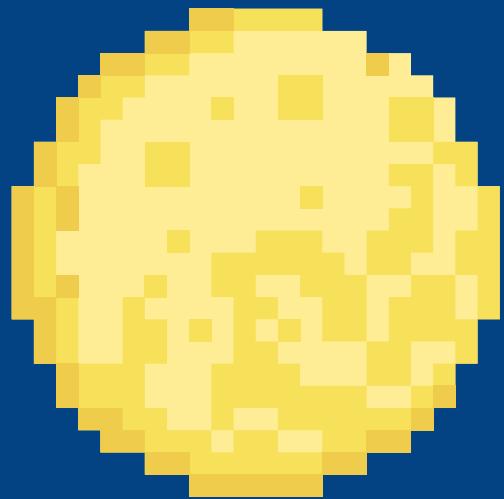
    private void OnKick(InputValue value) {
        anim.SetTrigger("kick");
    }

    private void FixedUpdate() {
        rb.linearVelocity = new Vector2(movement * speed, rb.linearVelocity.y);
        UpdateFacingDirection();
        UpdateAnimationState();
    }

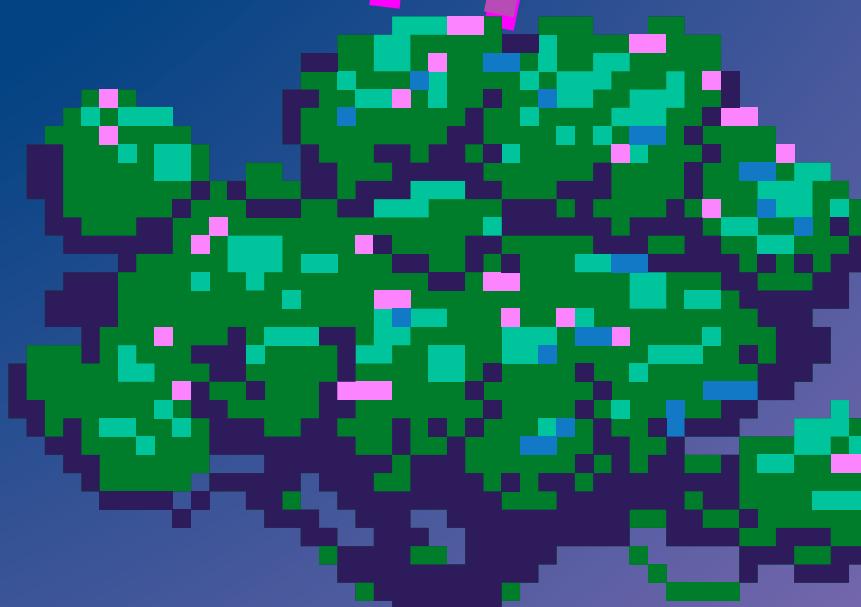
    private void UpdateAnimationState() {
        if (rb.linearVelocityY != 0 && !IsGrounded())
            anim.SetInteger("state", (int)AnimationState.Jumping);
        else if (rb.linearVelocityX == 0) {
            anim.SetInteger("state", (int)AnimationState.Idle);
        }
        else {
            anim.SetInteger("state", (int)AnimationState.Walking);
        }
    }

    private void UpdateFacingDirection() {
        if (movement != 0 && transform.localScale.x != movement)
            transform.localScale = new Vector3(transform.localScale.x * -1, transform.localScale.y, 0);
    }

    private bool IsGrounded() {
        return Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0, Vector2.down, 0.1f, groundLayer);
    }
}
```

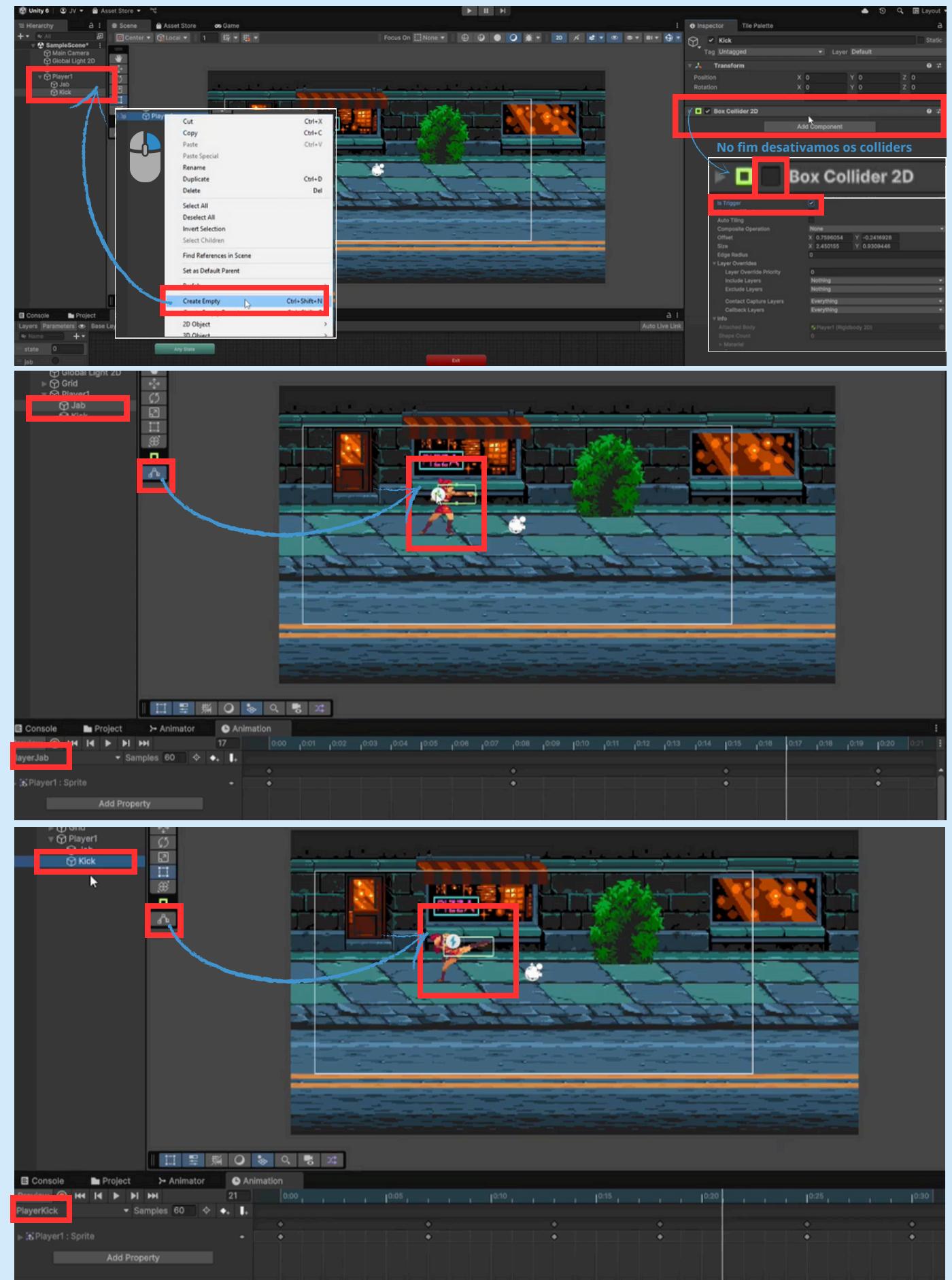


COMBAT



## Combate, Player 2

1- Para o combate, criamos dois Objects com uma “Box Collider 2D”. Ajustamos o collider de acordo com as animações e no fim selecionamos “is Trigger” e desativamos os colliders.



2- A seguir, criamos dois scripts, um script de “Attack” que adicionamos como componente para os ataques, neste caso, jab e kick, e outro script de “Damageable” para os players.

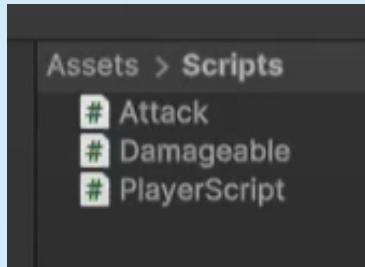
## Attack script

using UnityEngine;

```
public class Attack : MonoBehaviour
{
    [SerializeField] private int damage = 5;

    private void OnTriggerEnter2D(Collider2D collision) {
        Damageable enemy = collision.GetComponent<Damageable>();

        if (enemy != null)
            enemy.TakeDamage(damage);
    }
}
```



## Damageable script

using UnityEngine;

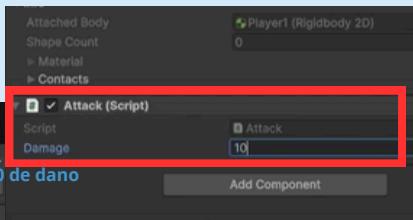
```
public class Damageable : MonoBehaviour
{
    [SerializeField] private int hp = 100;

    private Animator anim;

    private void Awake() {
        anim = GetComponent<Animator>();
    }

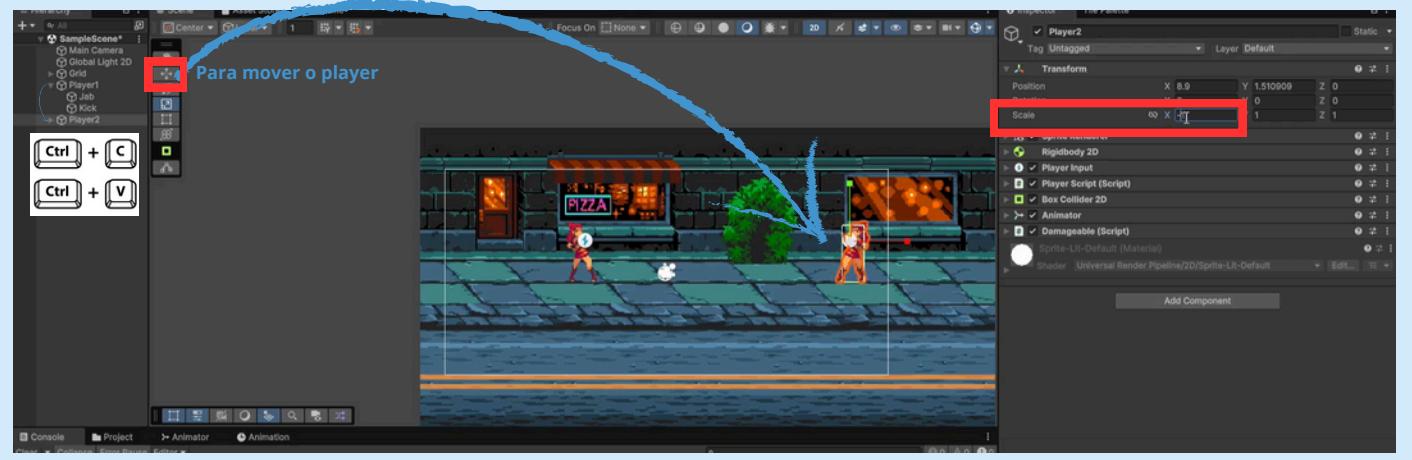
    public void TakeDamage(int damage) {
        hp -= damage;
        anim.SetTrigger("hurt");
    }
}
```

**ATENÇÃO:**  
Este script já contém animação para PlayerHurt, se tiveres dificuldades em fazer, visita a secção de animação ou o vídeo.

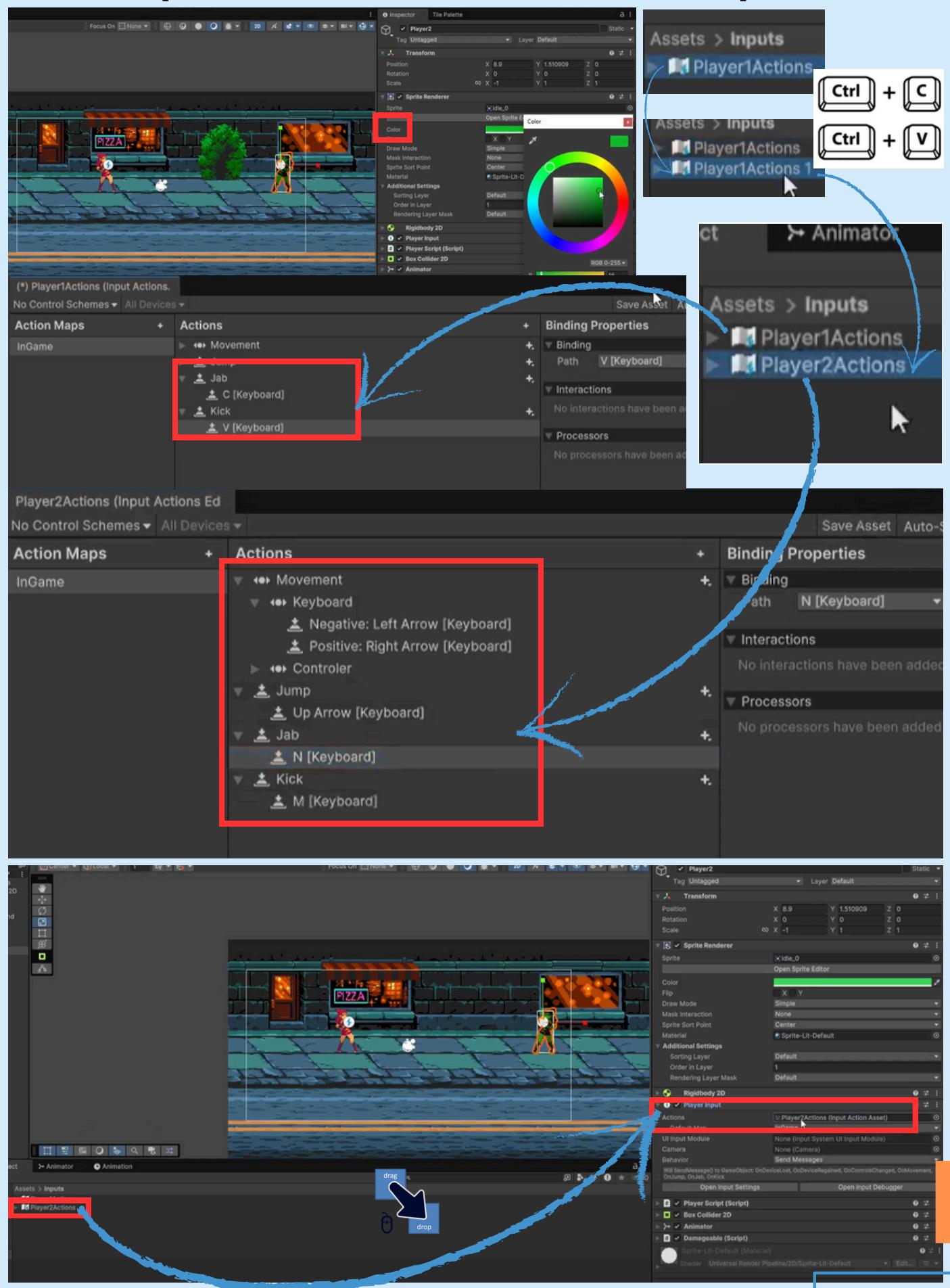


3- No fim, para ativarmos estes colliders quando pressionarmos as teclas, temos então de alterar a PlayerScript (script no fim da secção).

4- Para criarmos o segundo player e testar o combate, podemos simplesmente copiar e colar o Player1. Movemos o Player2 para o lado oposto e damos um scale -1 em x para virar a direção.



5- Para distinguir os players, podemos alterar a sua cor. Após isso, teremos de definir as teclas de input associadas às ações do Player 2. Neste caso, podemos copiar a Player1Actions, colar como “Player2Actions” e mudar as teclas associadas tal como na imagem abaixo.



## PlayerScript v3 with changes for combat:

```
using UnityEngine;
using UnityEngine.InputSystem;
using System.Collections;

public class PlayerScript : MonoBehaviour
{
    private float movement;
    [SerializeField] private float speed = 5;
    [SerializeField] private float jumpForce = 10;
    private enum AnimationState { Idle, Walking, Jumping, Falling, Kicking, Punching};

    private Rigidbody2D rb;
    private Collider2D coll;
    private Animator anim;
    private LayerMask groundLayer;
    private Collider2D jabHitbox;
    private Collider2D kickHitbox;

    private void Awake() {
        rb = GetComponent<Rigidbody2D>();
        coll = GetComponent<Collider2D>();
        anim = GetComponent<Animator>();
        groundLayer = LayerMask.GetMask("Ground");
        jabHitbox = transform.Find("Jab").GetComponent<Collider2D>();
        kickHitbox = transform.Find("Kick").GetComponent<Collider2D>();
    }

    ...

    private void OnJab(InputValue value) {
        anim.SetTrigger("jab");
        StartCoroutine(jab());
    }

    IEnumerator jab() {
        yield return new WaitForSeconds(0.08f);
        jabHitbox.enabled = true;
        yield return new WaitForSeconds(0.12f);
        jabHitbox.enabled = false;
    }

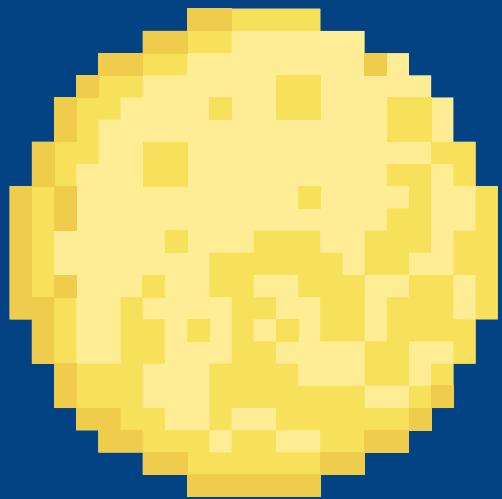
    private void OnKick(InputValue value) {
        anim.SetTrigger("kick");
        StartCoroutine(kick());
    }

    IEnumerator kick() {
        yield return new WaitForSeconds(0.18f);
        kickHitbox.enabled = true;
        yield return new WaitForSeconds(0.12f);
        kickHitbox.enabled = false;
    }

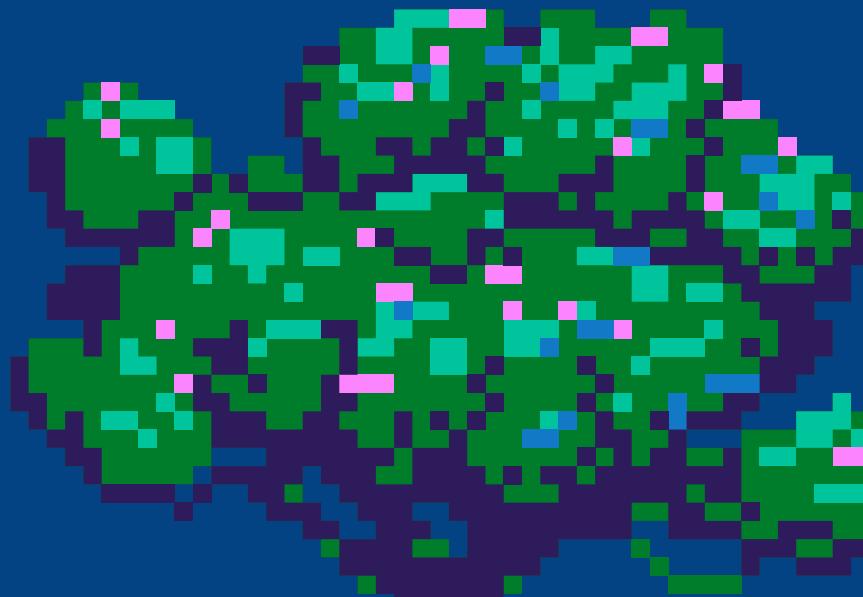
    private void FixedUpdate() {
        rb.linearVelocity = new Vector2(movement * speed, rb.linearVelocity.y);
        UpdateFacingDirection();
        UpdateAnimationState();
    }

    private void UpdateAnimationState() {
        if (rb.linearVelocityY != 0 && !IsGrounded())
            anim.SetInteger("state", (int)AnimationState.Jumping);
        else if (rb.linearVelocityX == 0) {
            anim.SetInteger("state", (int)AnimationState.Idle);
        }
        else {
            anim.SetInteger("state", (int)AnimationState.Walking);
        }
    }

    ...
}
```

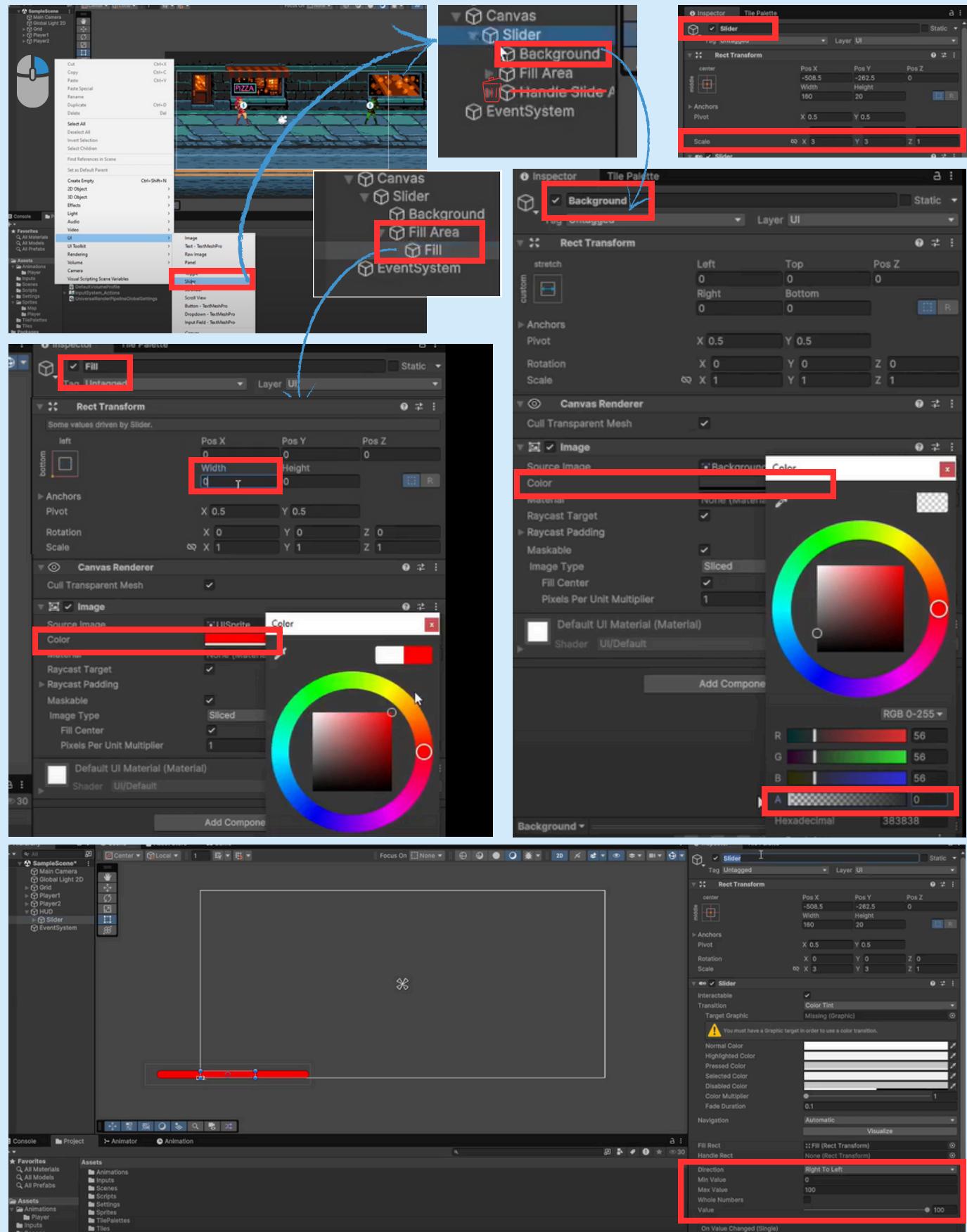


HUDO

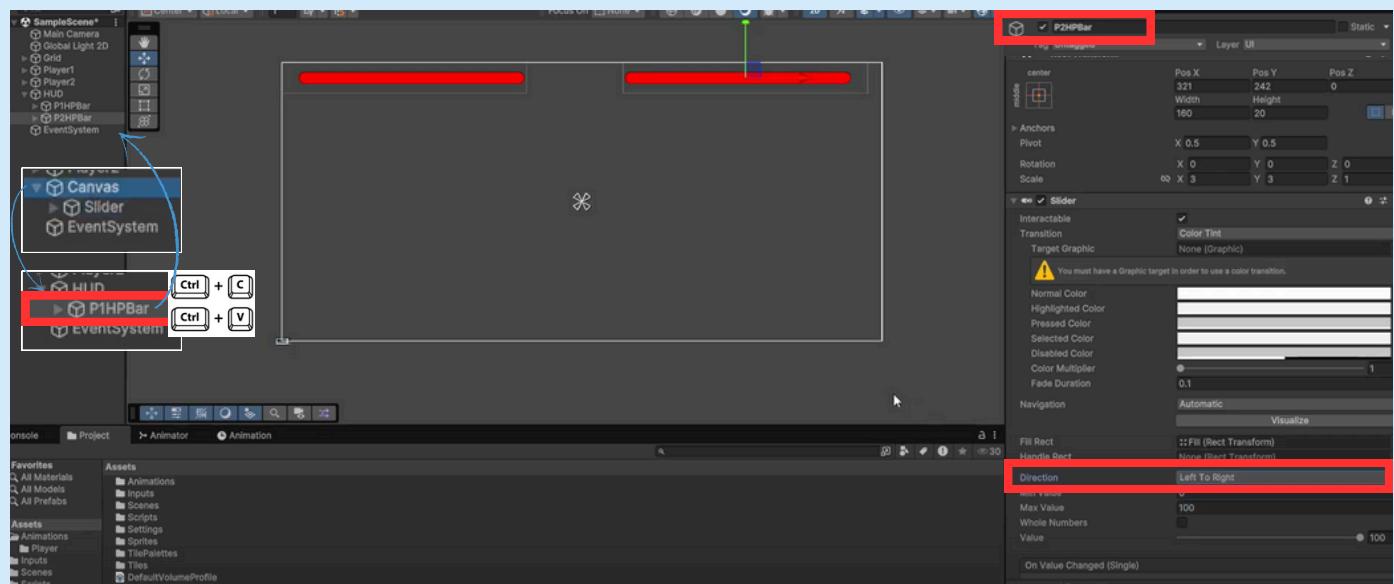


# HUD

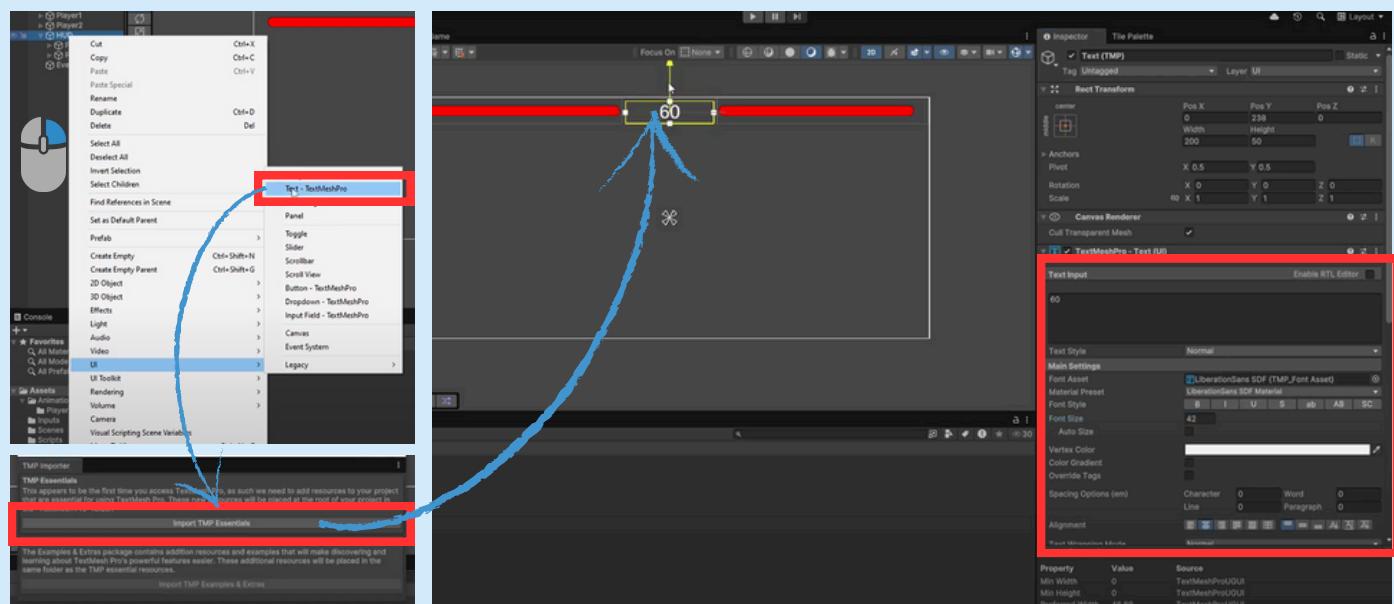
1- Para transmitirmos as informações do combate, criamos uma HUD. Começamos por criar um objeto do tipo UI>Slider. Damos um scale de 3 ao Slider e dentro dele, eliminamos o “Handle Slide”. A seguir, mudamos a cor de Background para transparente e Fill a vermelho com Width 0. No fim, alteraremos o Max Value, Value e Direction do Slider.



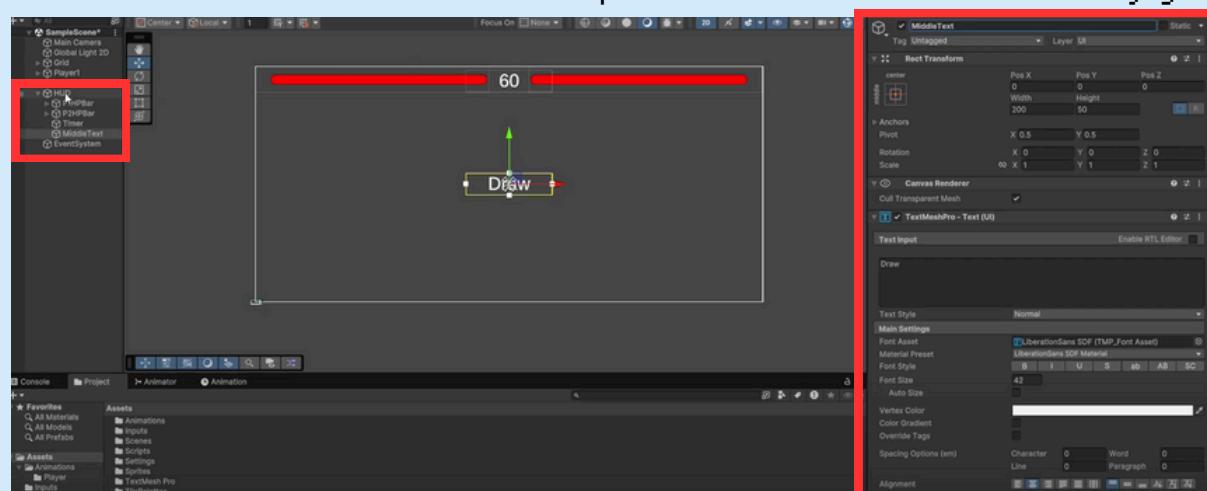
2- Após isso, podemos alterar o nome dos objetos por uma questão de organização e mover a P1HPBar para onde queremos dentro do canvas (tal como a imagem). Para o P2, copiamos e colamos no cenário e alteramos a Direction desta nova barra P2HPBar.



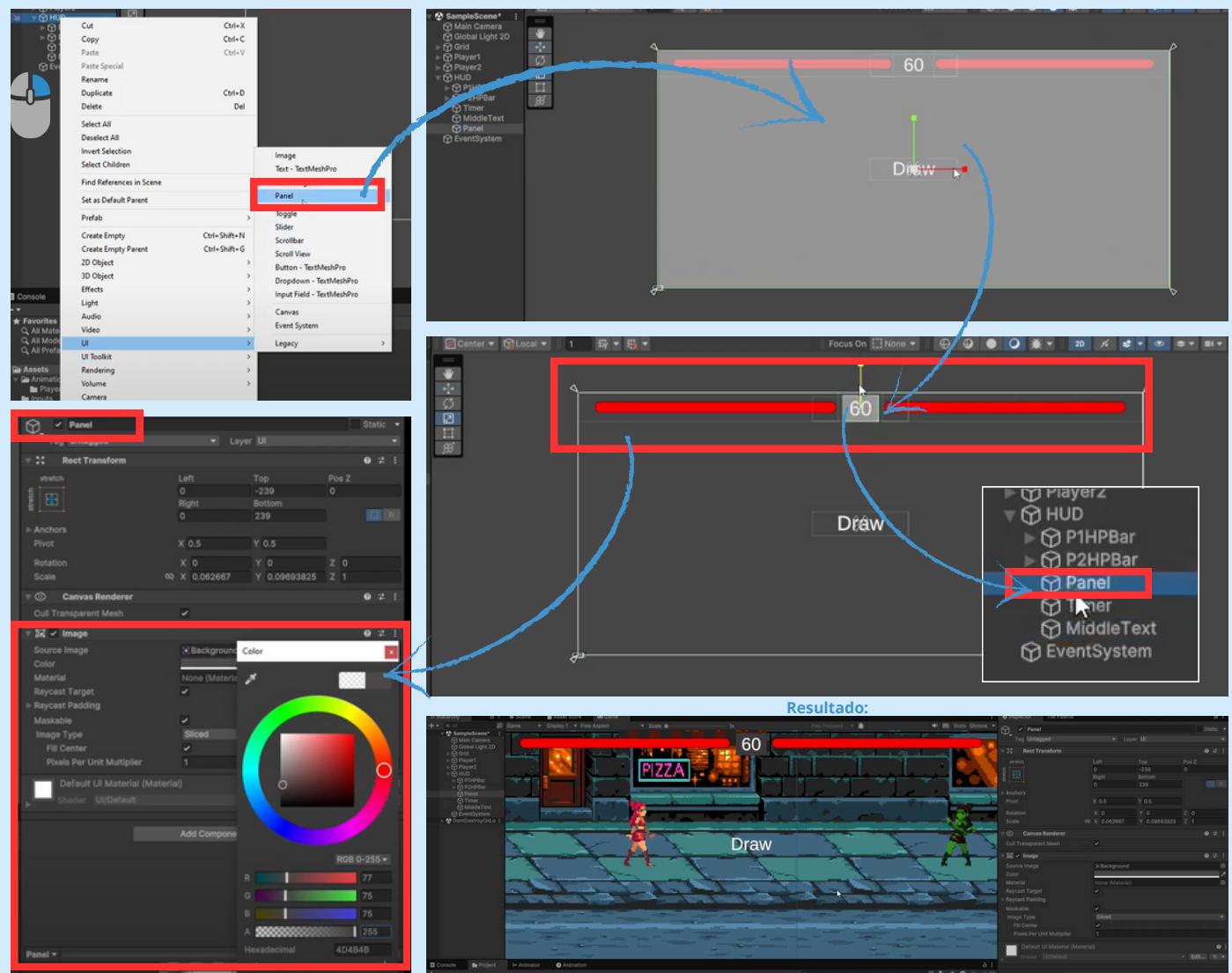
3- No caso de querermos um Timer para o jogo, criamos um novo UI Object dentro da HUD que é o TextMeshPro. Damos import da package e vai aparecer um texto onde podemos brincar com as propriedades dela como a font size, font style, color, alinhamentos e o texto em si.



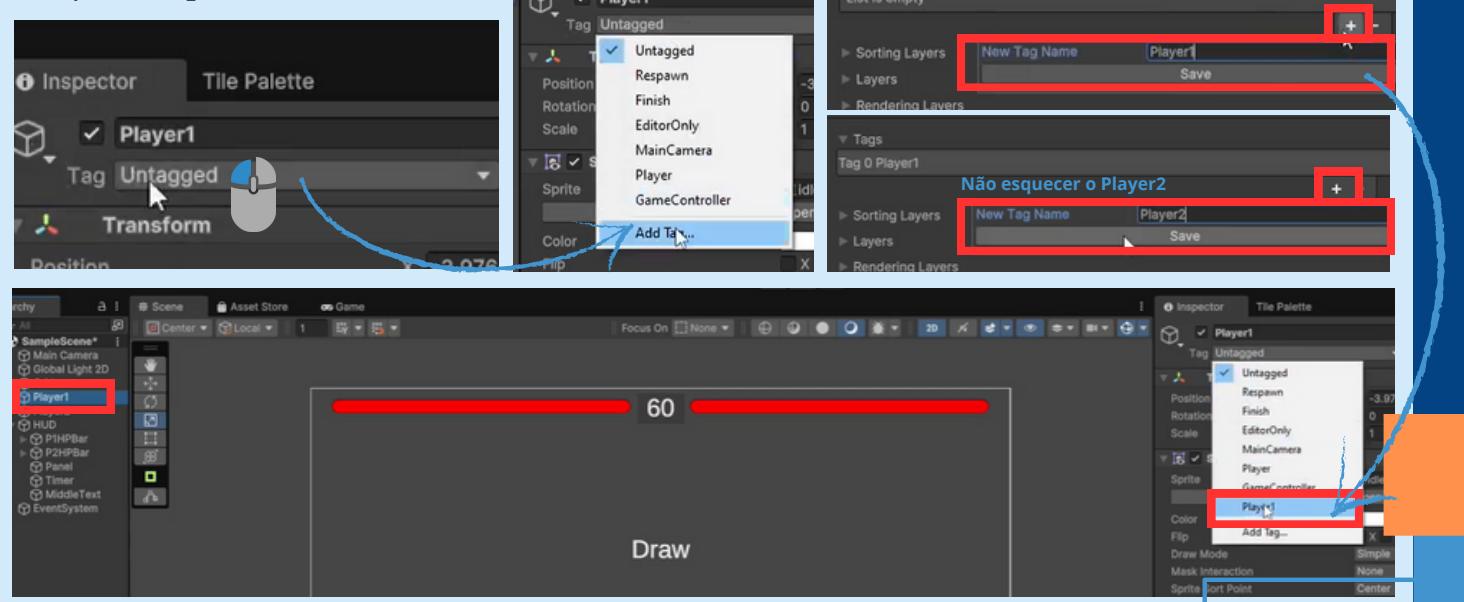
4- Queremos também criar outro TextMeshPro para declararmos o Winner no fim do jogo.



5- Para garantirmos visibilidade do timer, podemos criar um Panel com uma cor cinzenta mais opaco, por trás do texto do Timer. É preciso dar scale e mover este Panel e garantir que temos o Panel atrás do Timer.



6-No final, é necessário darmos uma Tag ao Player1 e também ao Player2. Para tudo funcionar, basta adicionarmos um script como uma componente ao HUD chamado HUDManager e modificar o script Damageable.



## HUDManager

```
using System;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using UnityEngine.SceneManagement;
using System.Collections;

public class HUDManager : MonoBehaviour
{
    private Slider p1HPBar;
    private Slider p2HPBar;
    private TMP_Text middleText;
    private TMP_Text timer;

    private float time = 60;

    private void Awake()
    {
        p1HPBar = transform.Find("P1HPBar").GetComponent<Slider>();
        p2HPBar = transform.Find("P2HPBar").GetComponent<Slider>();
        middleText = transform.Find("MiddleText").GetComponent<TMP_Text>();
        timer = transform.Find("Timer").GetComponent<TMP_Text>();
    }

    private void Update()
    {
        time -= Time.deltaTime;

        timer.text = ((int)time).ToString();

        if (time <= 0)
        {
            ShowMiddleText();
        }
    }

    private void OnEnable()
    {
        Damageable.OnDamageTaken += DecreaseHPBar;
    }

    private void OnDisable()
    {
        Damageable.OnDamageTaken -= DecreaseHPBar;
    }

    private void DecreaseHPBar(float hp, String tag)
    {
        if (tag == "Player1")
            p1HPBar.value = hp;
        else
            p2HPBar.value = hp;

        if (hp <= 0)
            ShowMiddleText();
    }

    private void ShowMiddleText()
    {
        middleText.enabled = true;
        if (p1HPBar.value < p2HPBar.value)
            middleText.text = "P2 Wins!";
        else if (p1HPBar.value > p2HPBar.value)
            middleText.text = "P1 Wins!";
        else
            middleText.text = "Draw!";

        StartCoroutine(endGame());
    }
}
```

## Damageable v2

```
using System;
using UnityEngine;

public class Damageable : MonoBehaviour
{
    public delegate void DamgeTakenAction(float hp, String tag);
    public static event DamgeTakenAction OnDamageTaken;

    [SerializeField] private int hp = 100;

    private Animator anim;

    private void Awake()
    {
        anim = GetComponent<Animator>();
    }

    public void TakeDamage(int damage)
    {
        hp -= damage;
        anim.SetTrigger("hurt");
        OnDamageTaken?.Invoke(hp, gameObject.tag);
    }
}
```



Chegaste ao fim deste **FIGHTER GAME TUTORIAL!**



Na playlist do vídeo, ainda existe um vídeo onde ensinamos

**MENU** [como criar um Menu Inicial!](#)

Recomendamos fortemente que vejam os vídeos pois incluem explicações detalhadas sobre o código.

