Game Development Interactive Tutorial | Free Version

Table of Contents:

💡 Tip: For the best learning experience, follow along with the Video tutorials as you go. Hands-on practice will help solidify what you're learning!

0. Introduction

Welcome to Your Game Development Journey!

In this blog, you'll learn everything you need to build your first game step-by-step. This guide is made specifically for those new to game development, offering clear, beginner-friendly explanations. Each post breaks down crucial coding steps, supported by video tutorials, so you can follow along visually. By the end, you'll not only have a working game but also the skills to keep building more on your own!

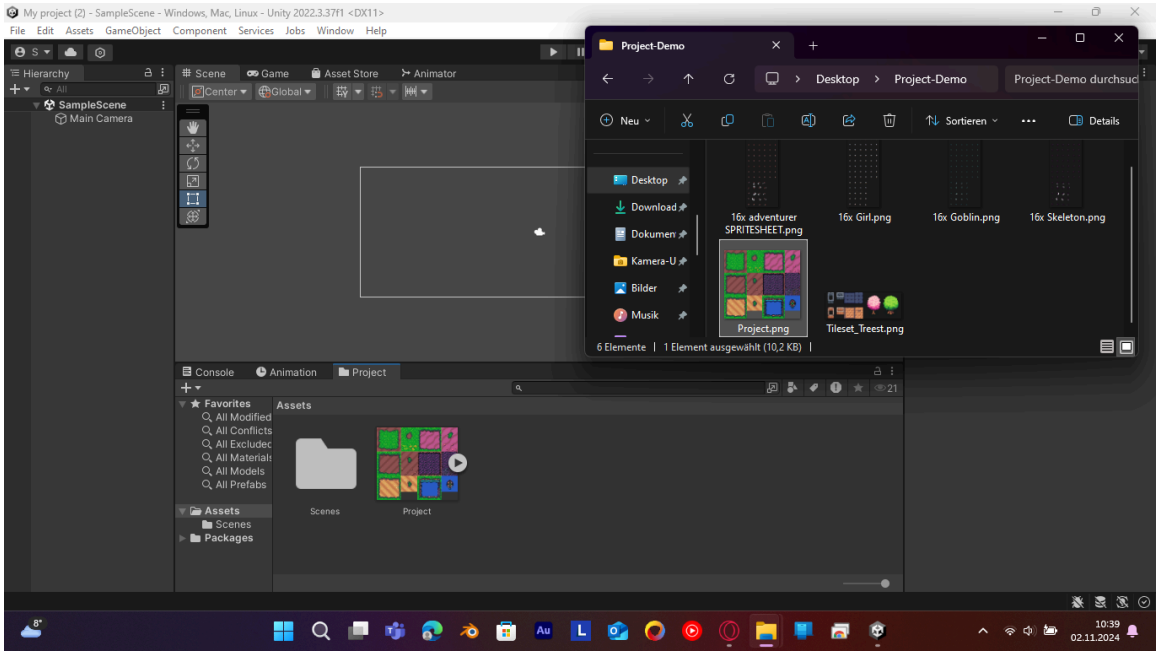
Before we dive in, take a moment to answer these questions to help you get the most out of this journey! Reply below with your answers so we can support you better:

https://smoggy-bill-915.notion.site/1313582bd35c803a95aace84d8ade416?pvs=105

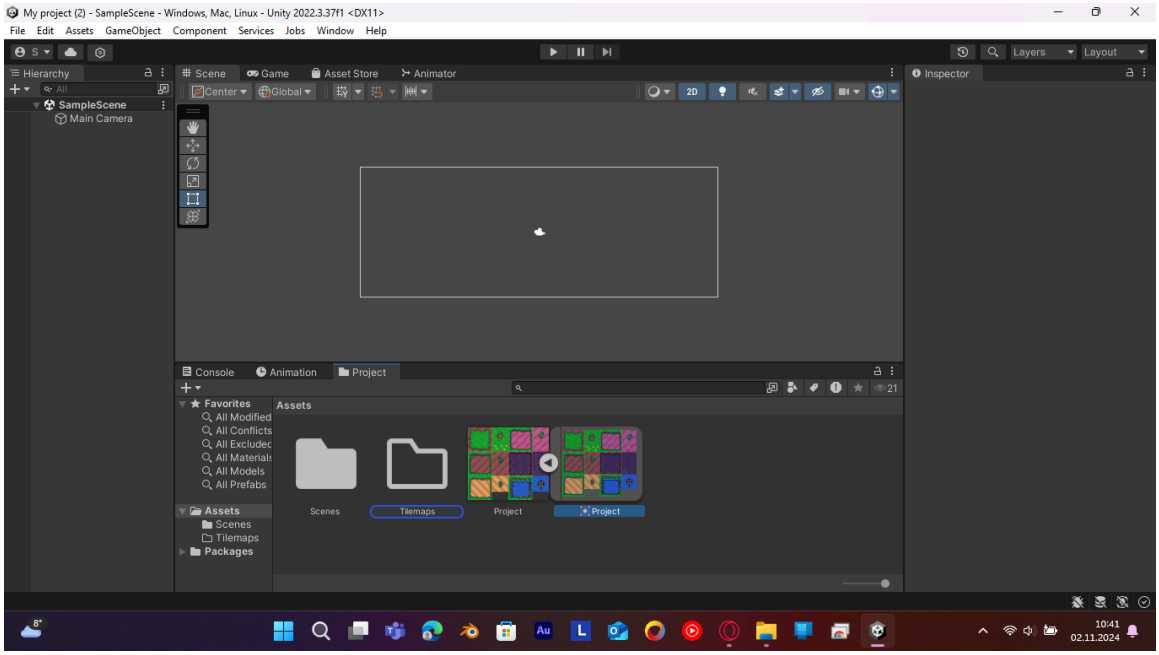
1. Tilemaps

The first step after creating a new 2D-Project, is building Tilemaps, where our Players and Enemies can move.

In order to do that, **1. Open your project → 2. Drag and Drop the Tiles you want to add from your Pixel Art Bundle into Assets**



→ 3. Create a new Folder in Assets and name it “Tilemaps” and move the Tiles into that Folder



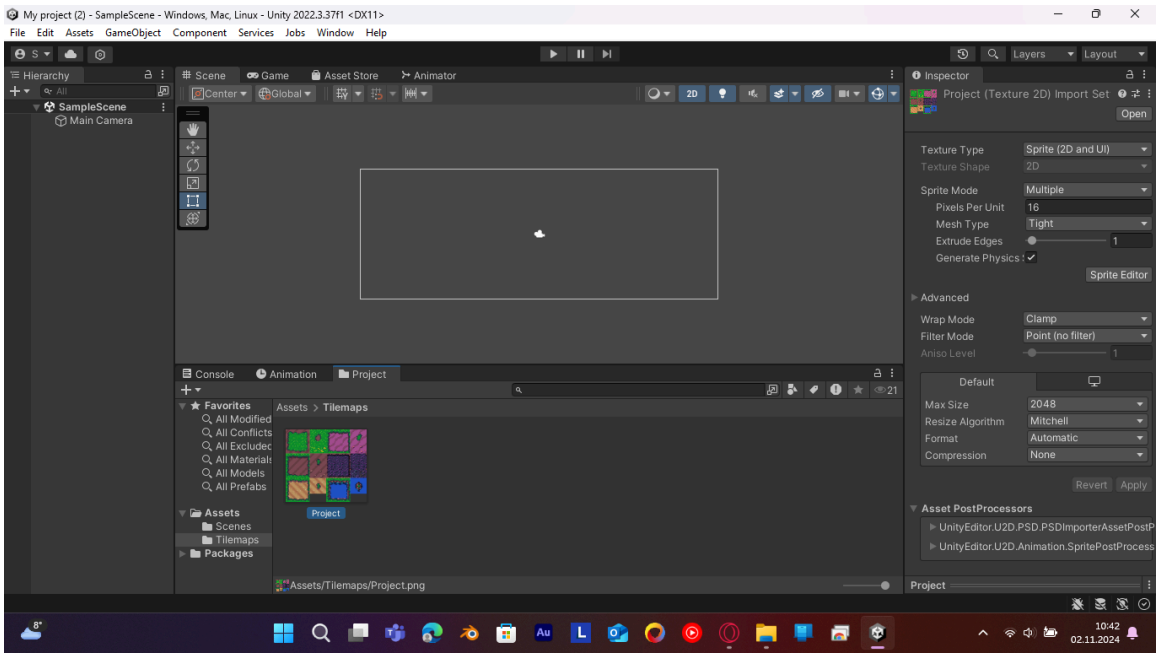
→ 4. Click on your Tiles and make sure you change the following settings:

Sprite Mode: **Multiple**

Pixels per Unit: **16**

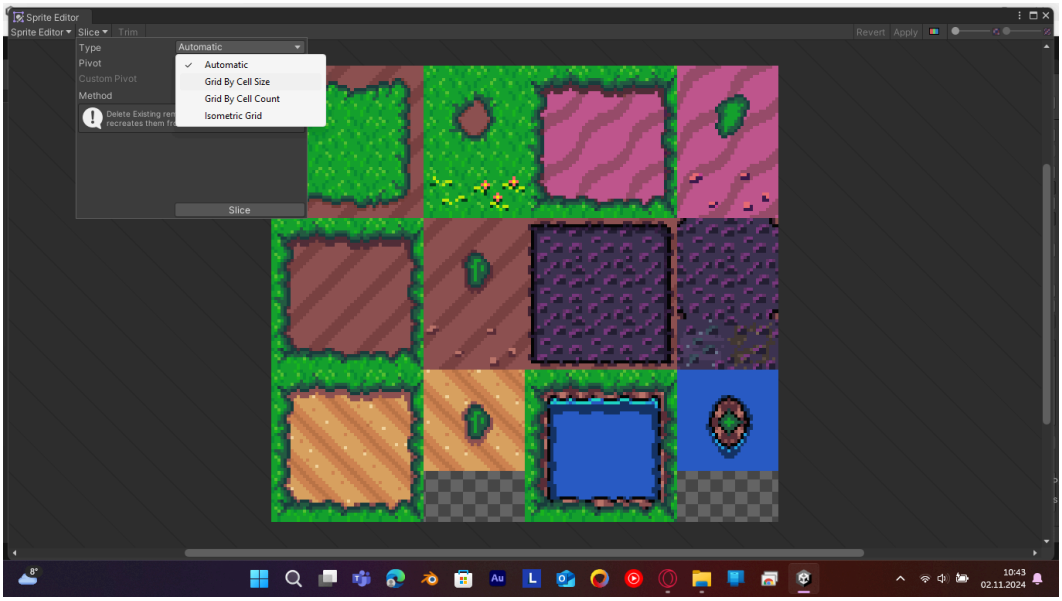
Filter Mode: **Point (No Filter)**

Compression: **None**

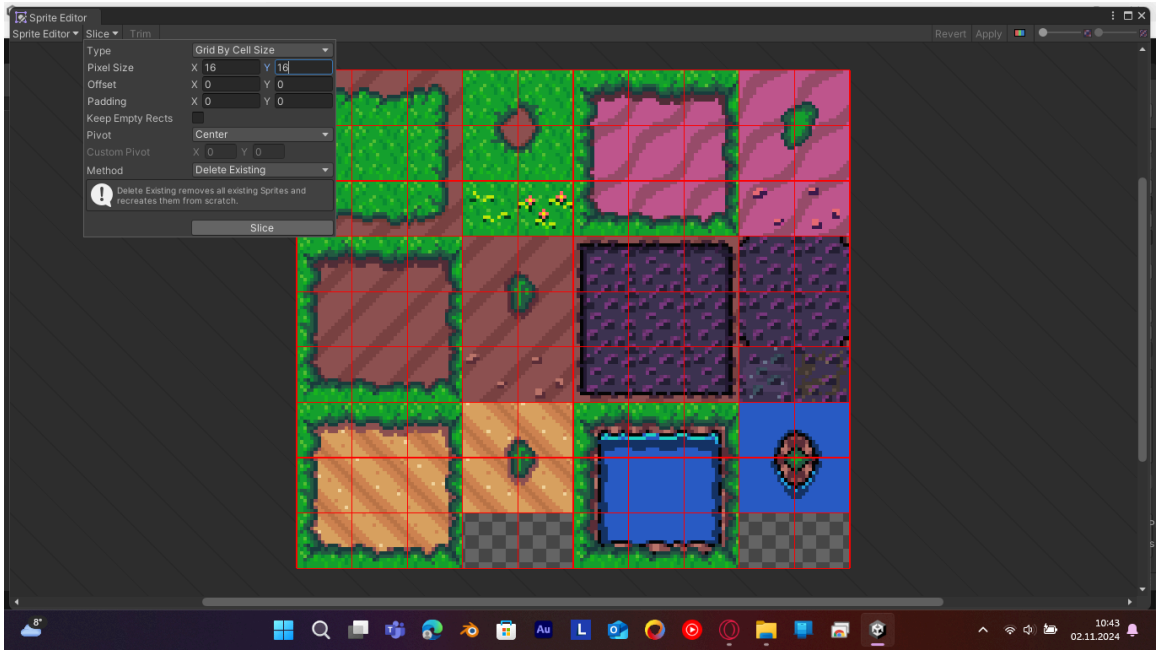


After that don't forget to hit **apply** and then open the Sprite editor.

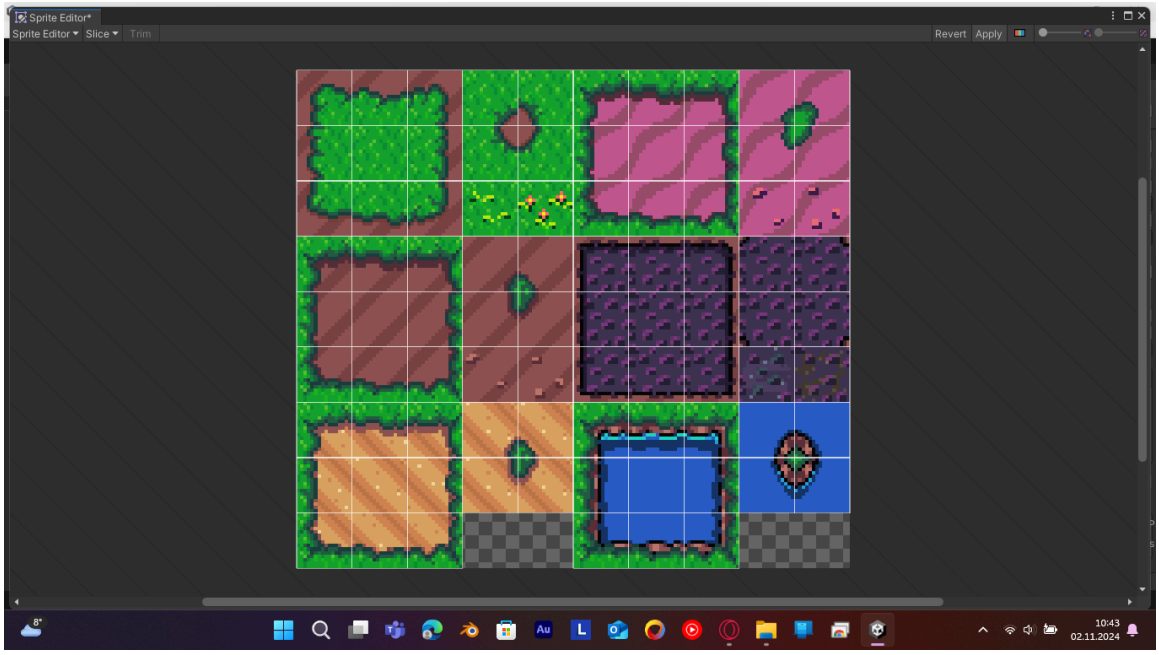
5. **In here click on Slice** → 6. **Grid by Cell Size**



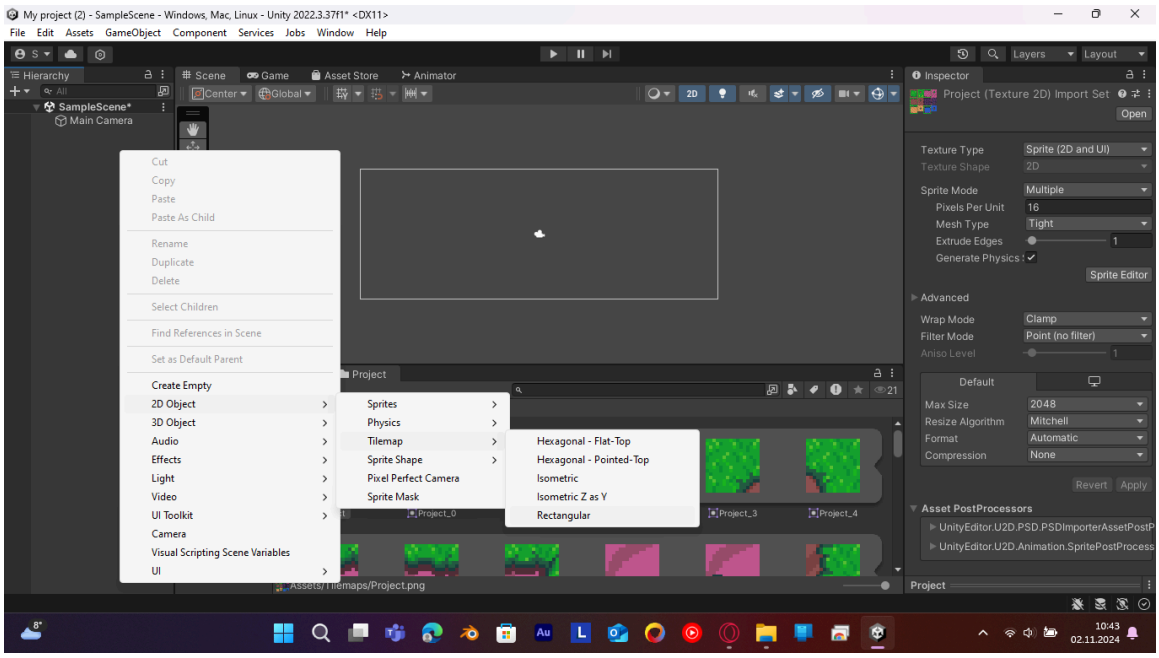
7. **Write 16 × 16 for Pixel Size, since we're dealing with Tiles that are** 16×16 → 8. **Click on Slice**



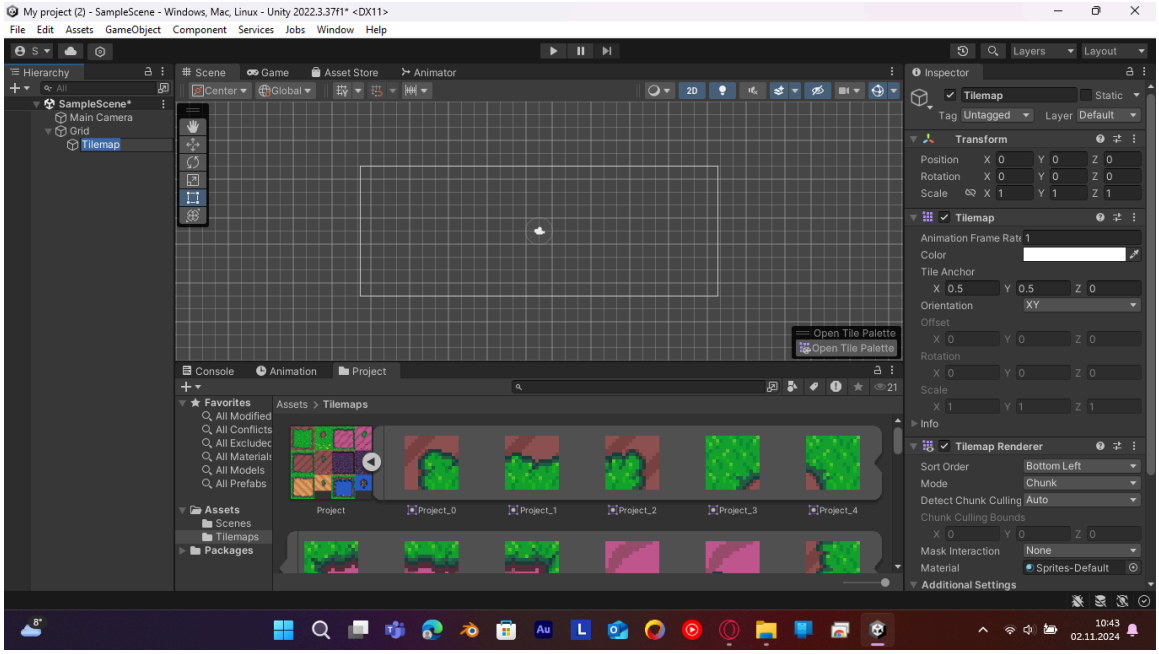
9. **Hit Apply**



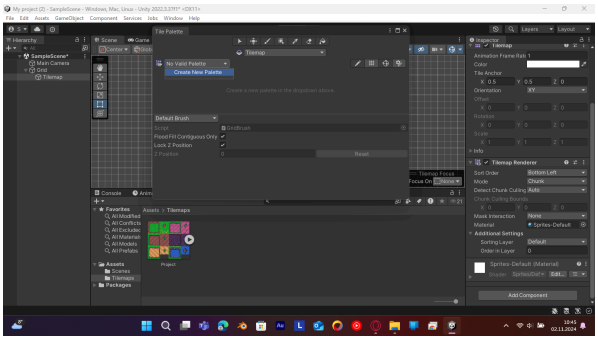
10. **Under the Hierarchy section → Right click → 2D Object → Tilemap → Rectangular**

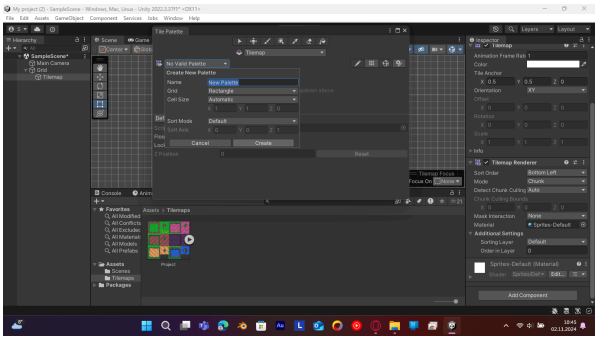


11. **Call it Tilemap and then open Tile Palette**

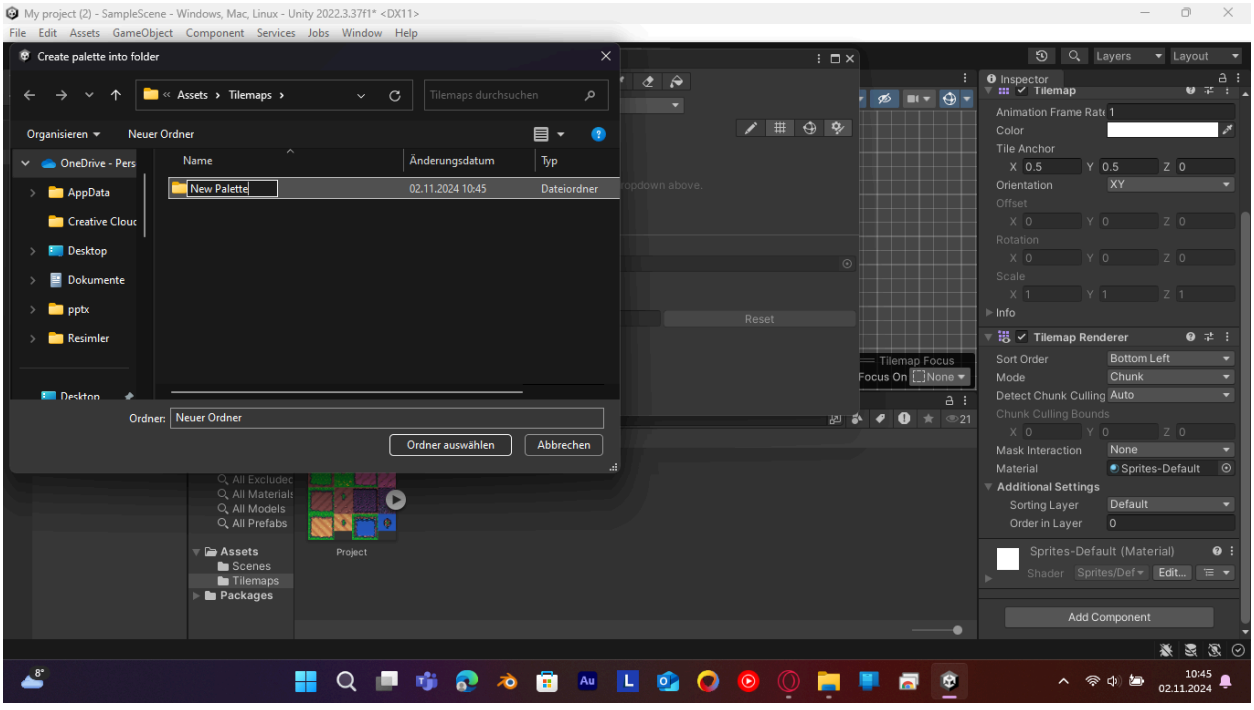


12. **Create a New Palette**

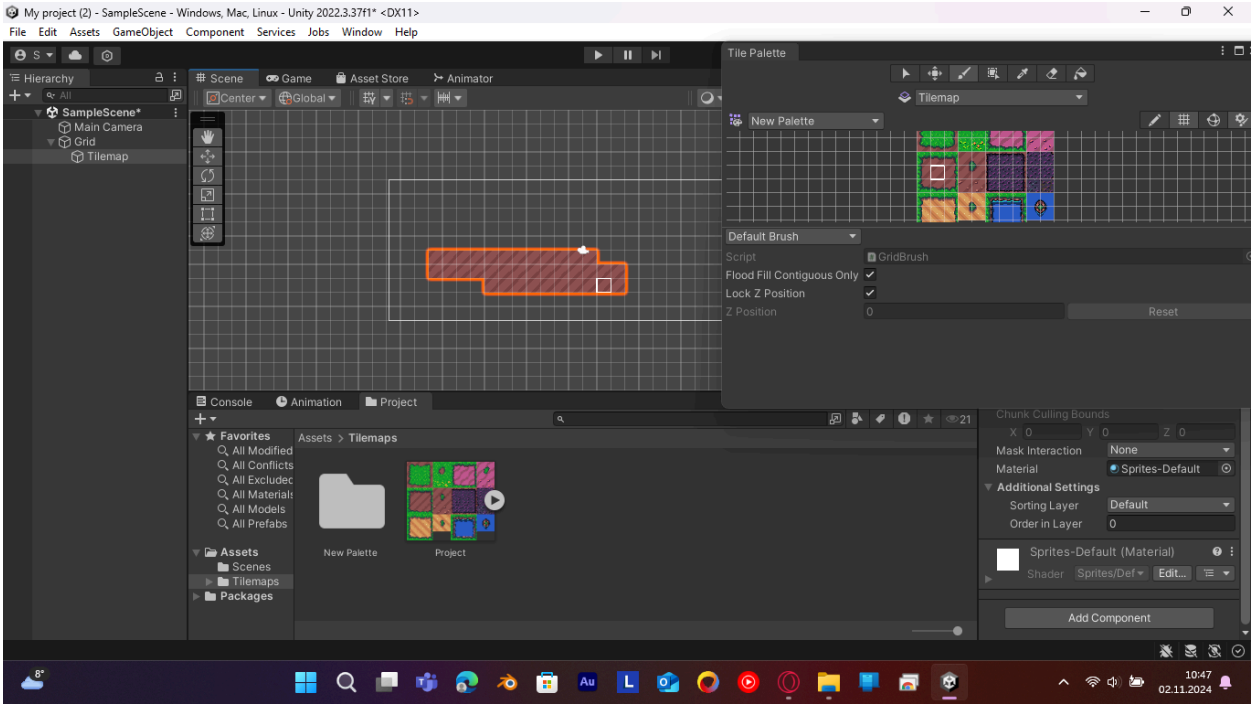




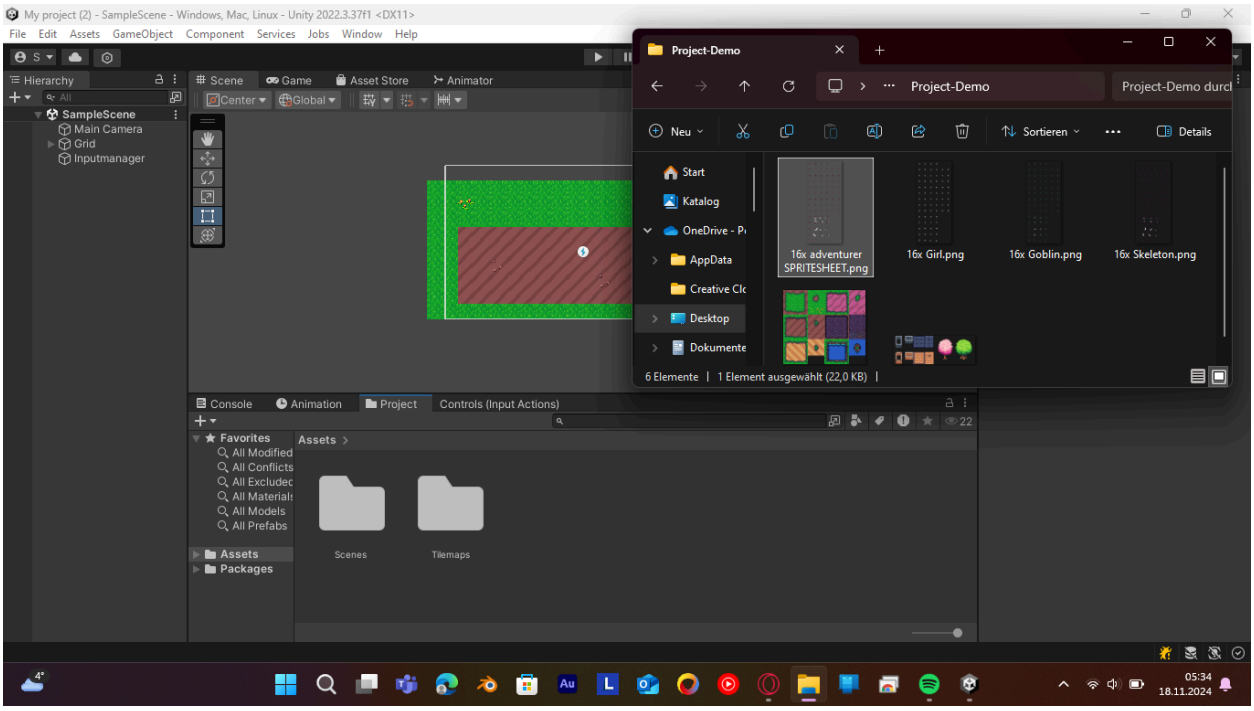
13. Create a new File and call it whatever you want → choose the Folder you just created.



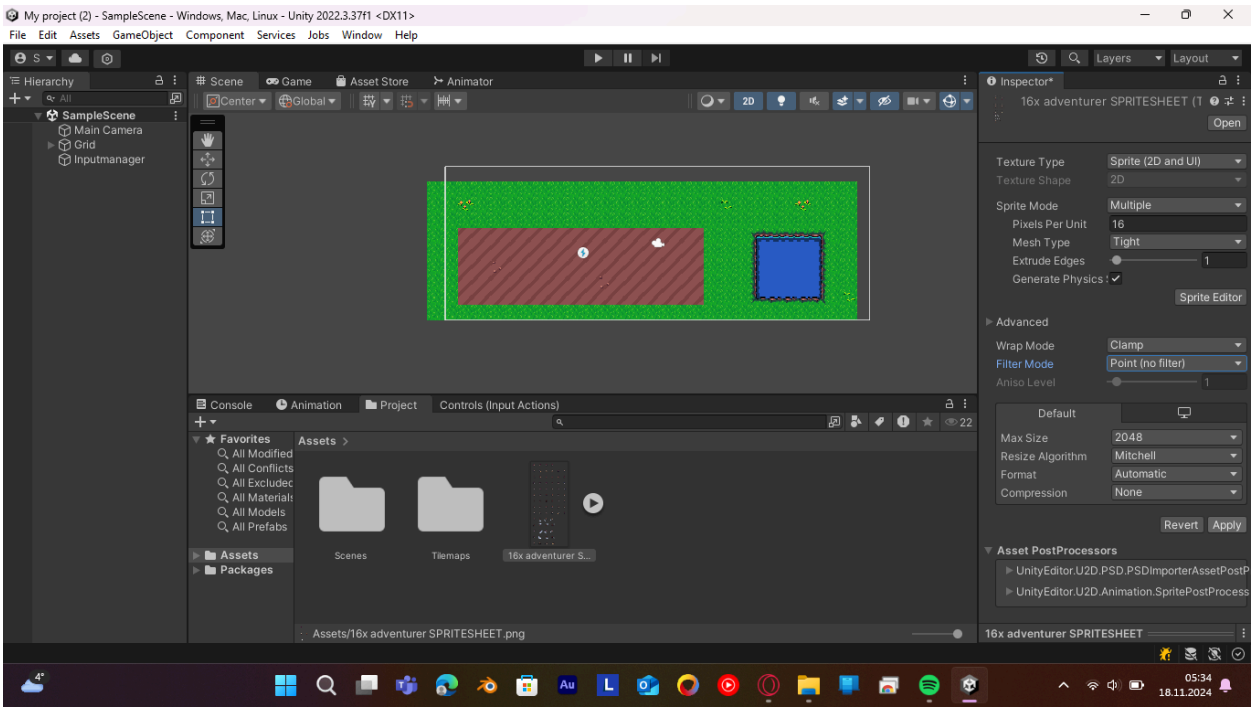
14. **Take the Paint-Brush-Tool and select an Area you would like to draw → Make sure you have Tilemap selected and then start drawing into your scene.**



Mini Step: Adding the Character to our scene



The next mini-step will be adding our character to the scene, click on your sprite sheet and make sure you have the following settings:



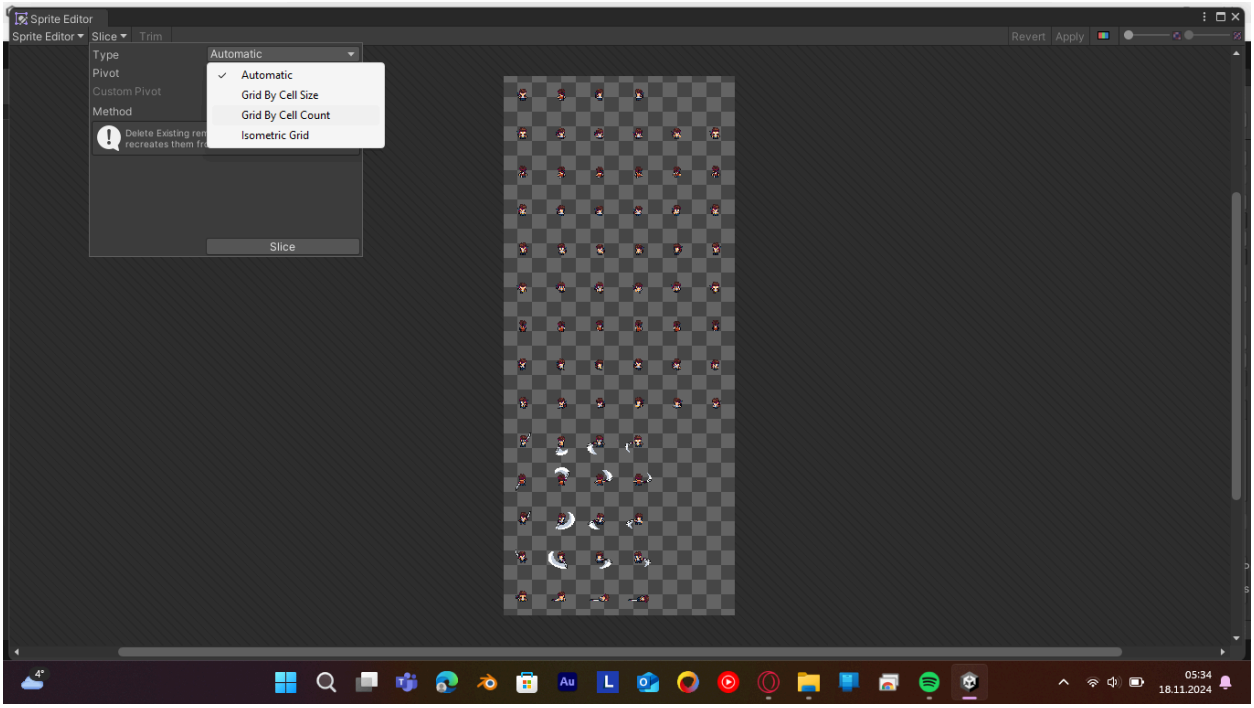
Sprite Mode: **Multiple**

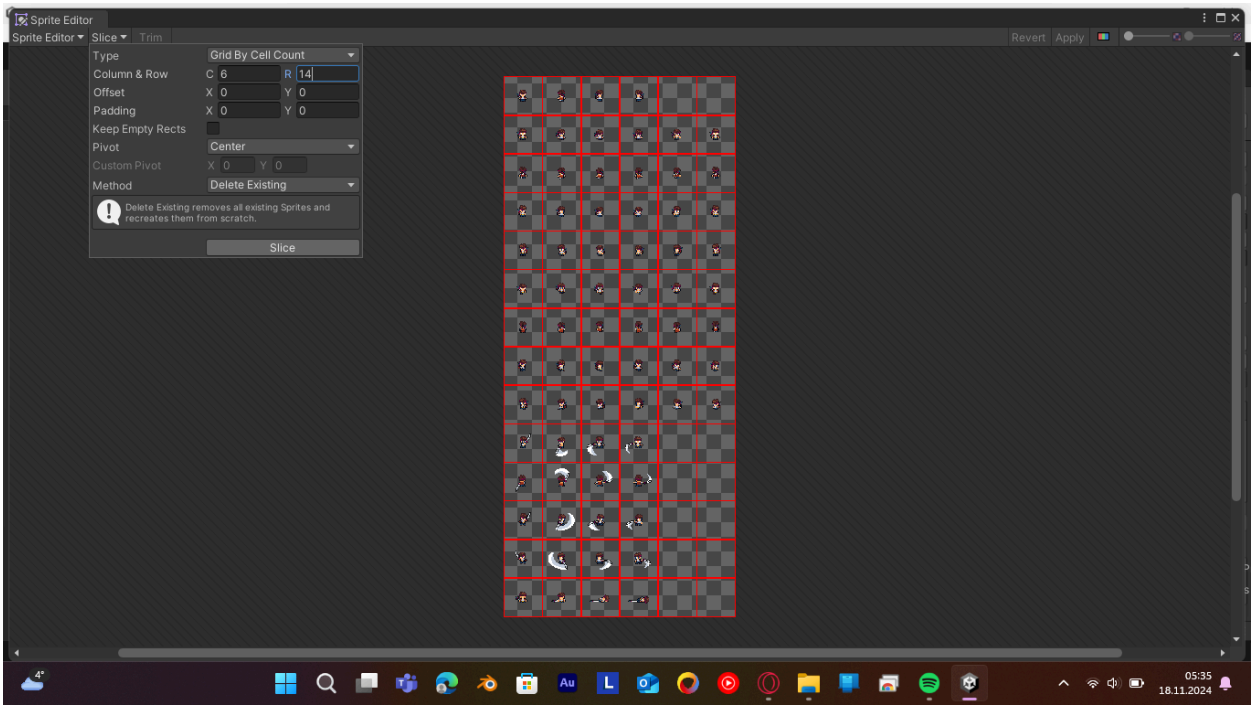
Pixels per Unit: **16**

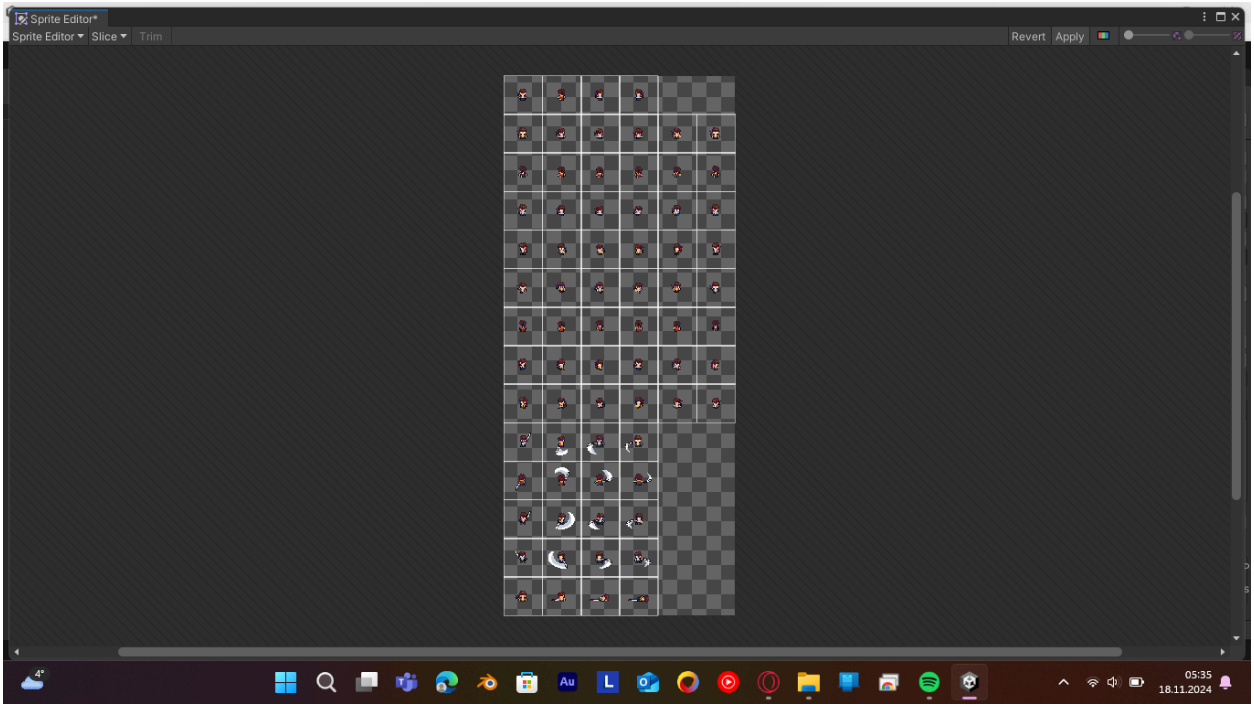
Filter Mode: **Point (No Filter)**

Compression: **None**

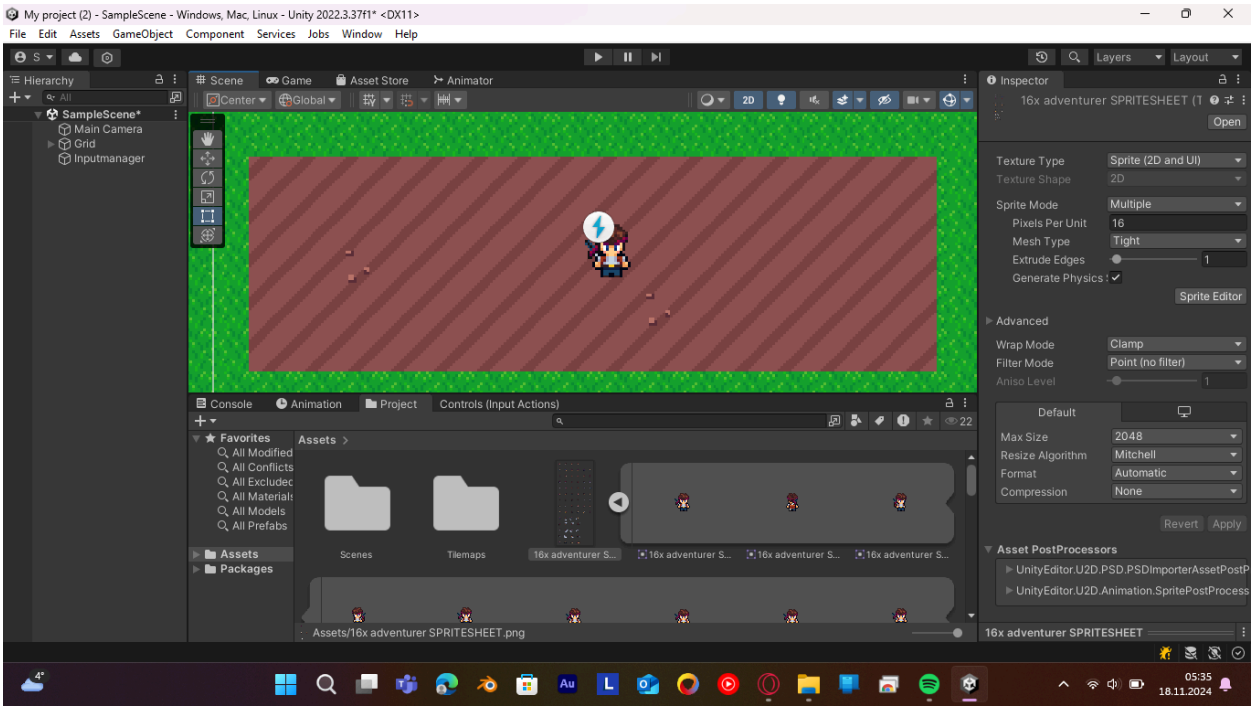
Hit **Apply,** open your sprite editor → click on slice first → grid by cell count → 6:14 ratio → slice → apply.


Sprite Editor
Sprite Editor
Slice
Trim
Revert
Apply
Type
Automatic
Pivot
Custom Pivot
Method
Automatic
Grid By Cell Size
Grid By Cell Count
Isometric Grid
Slice
05:34
18.11.2024


Sprite Editor
Sprite Editor
Slice
Trim
Revert
Apply
Type
Grid By Cell Count
Column & Row
C 6
R 14
Offset
X 0
Y 0
Padding
X 0
Y 0
Keep Empty Rects
Pivot
Center
Custom Pivot
X 0
Y 0
Method
Delete Existing
Delete Existing removes all existing Sprites and recreates them from scratch.
Slice
05:35
18.11.2024



You can now drag and drop your characters basic look into your scene



2. Basic Movement

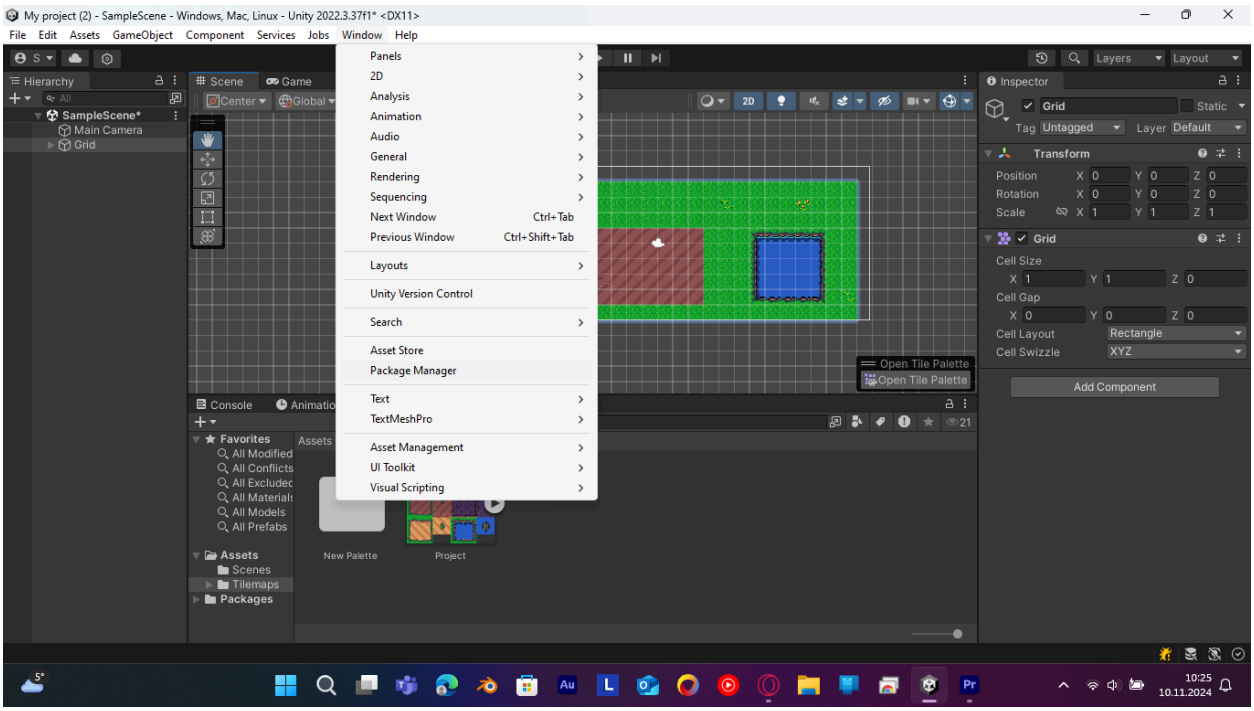
Welcome to the **Movement** Section of this Tutorial! If you've made it here after creating your first tileset, you're already off to an incredible start! I want to give you major props—designing that first asset is no small feat. Honestly, when I first tried to create my own tileset, I couldn't even finish it and ended up stepping back from pixel art for a while. It felt overwhelming and, I thought maybe it just wasn't for me. But here you are, pushing through, and I'm really glad you're taking it one step at a time.

In this section, we're focusing on movement, one of the most fundamental mechanics that brings a game to life. Basic player movement will be the first dynamic element you add, and it's where all the pieces start coming together. As you go through the tutorials and implement the code, you'll see how each detail—from speed settings to direction—creates the foundation for all future features.

2.1 Unity Input System

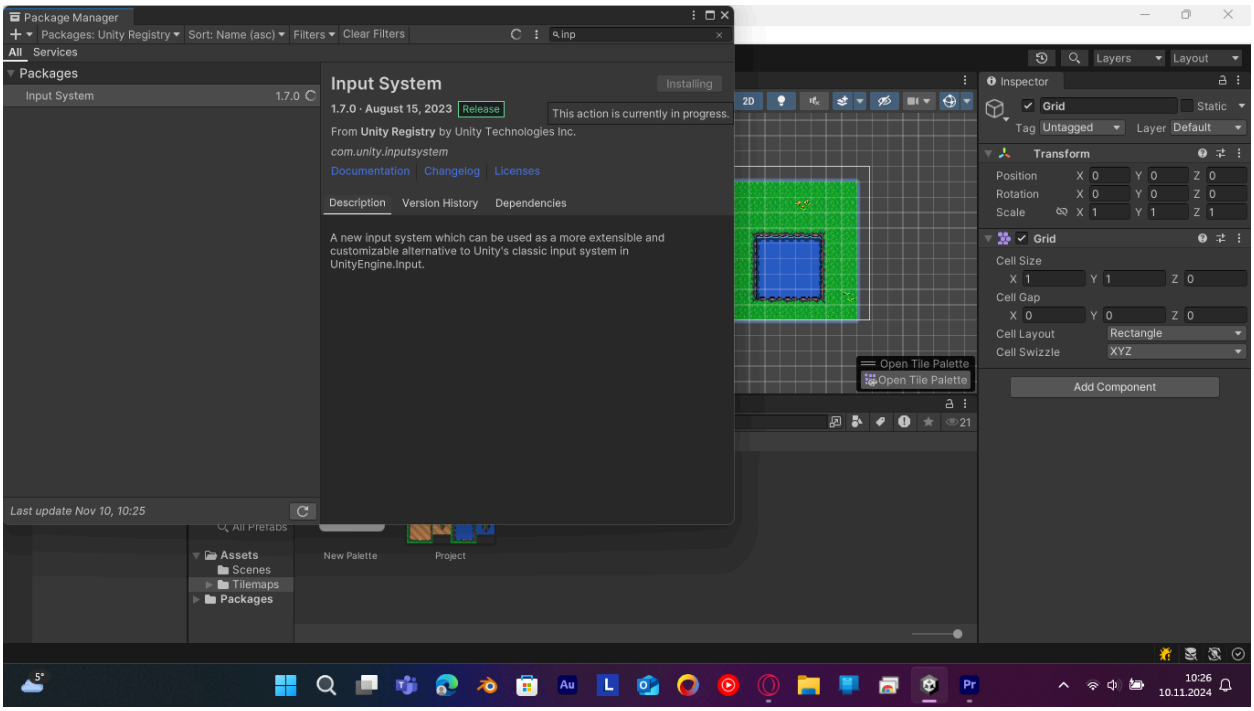
We'll begin by exporting our character sprite and slicing it to prepare for smooth animations, just like we did with our Tilesets. Then, we'll create our 2D movement using Unity's *New Input System*, which makes everything a whole lot easier and more flexible for our project. To get started, we'll download and install the New Input System package

1. Under Window open → Package Manager

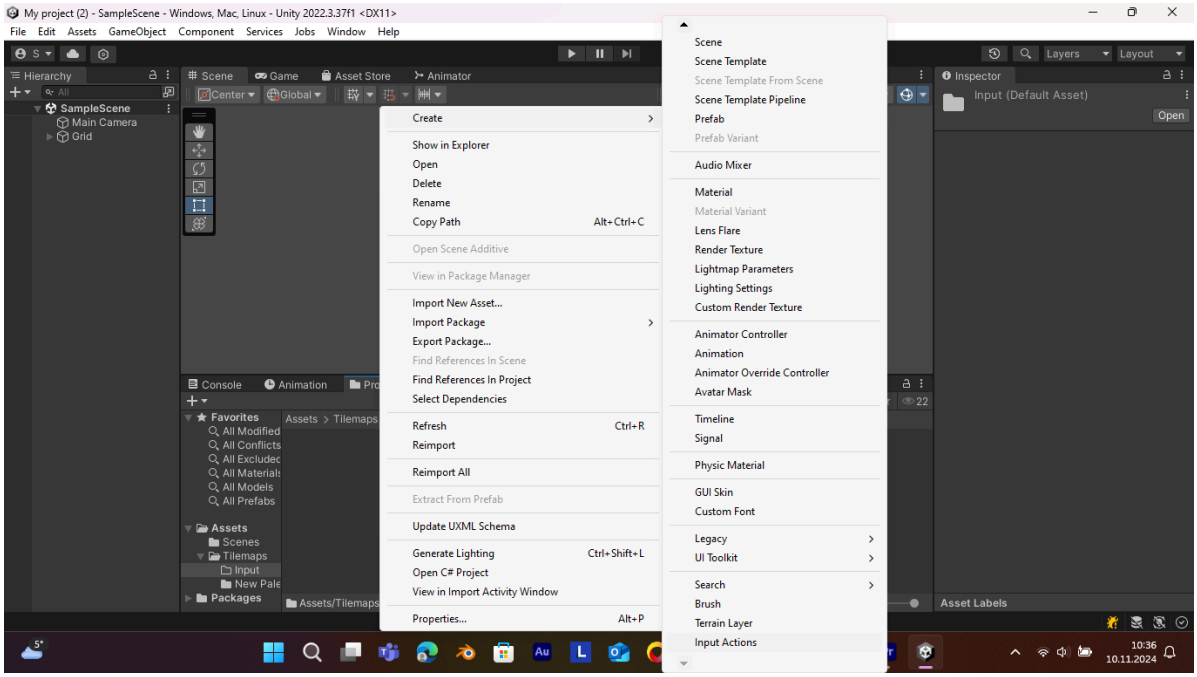


2. Select Packages:

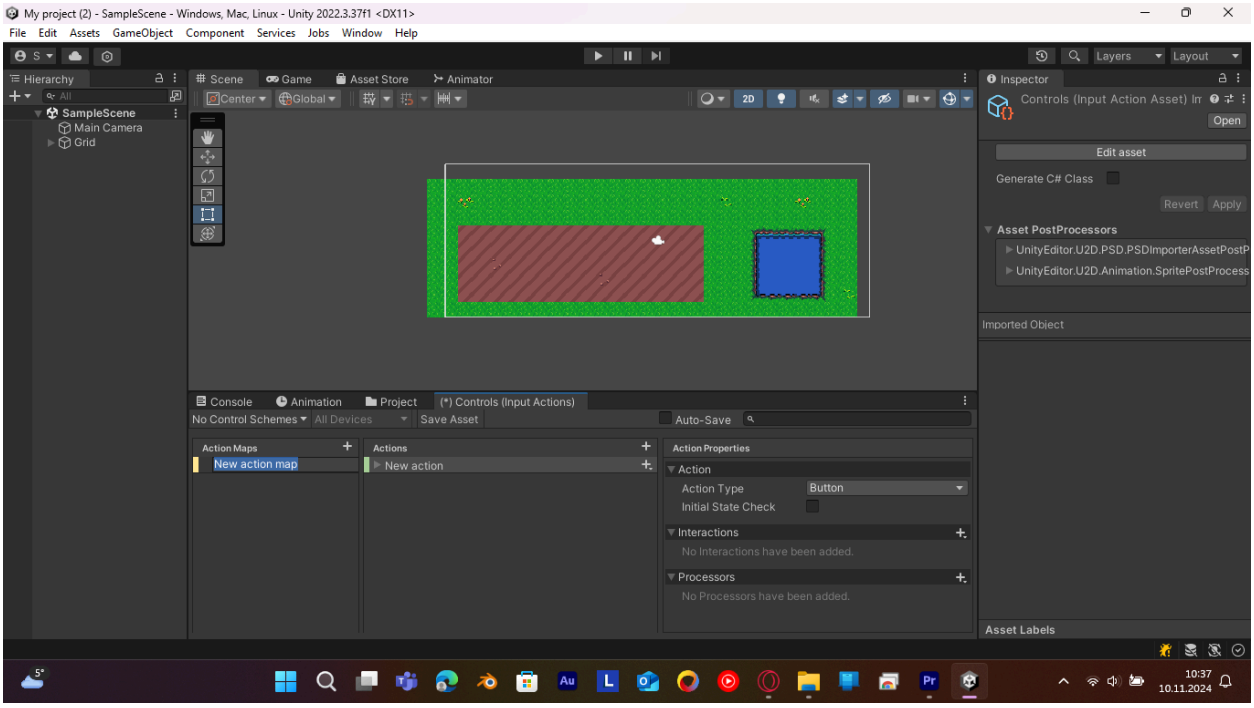
Unity Registry → search for Input System and download it



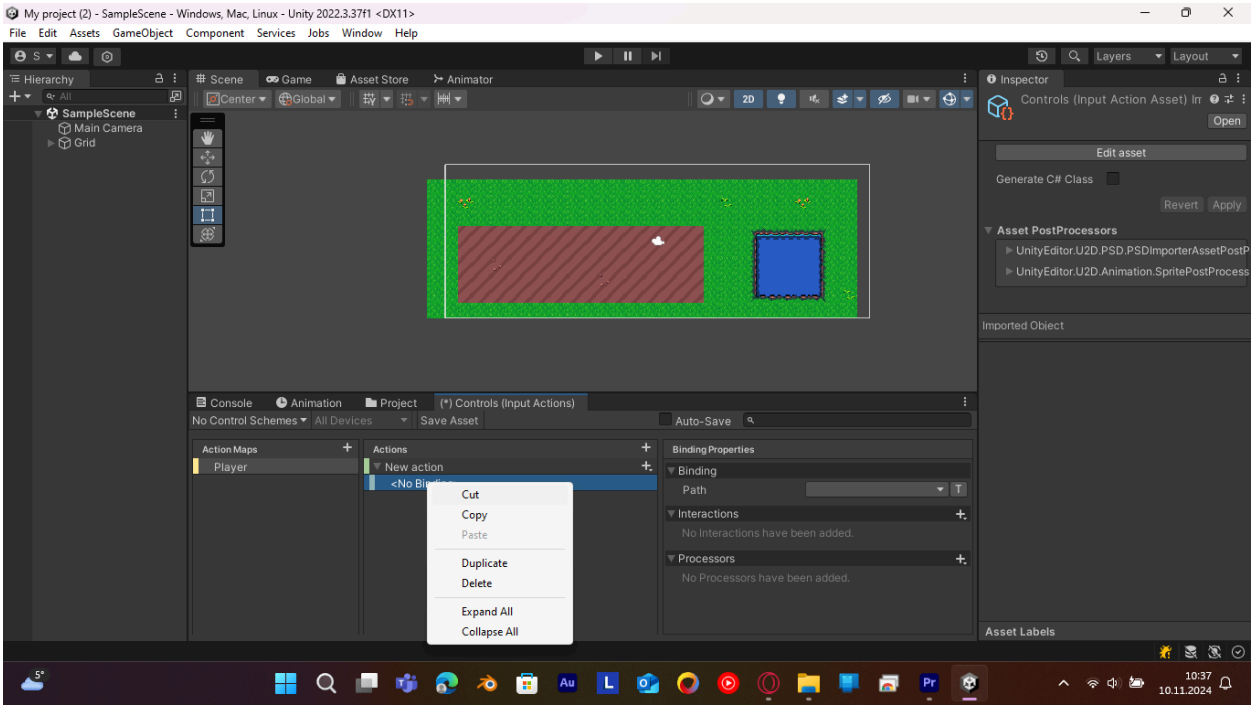
3. Create a new **Folder** (call it: Input) → **Right click** → **Create** → **New Input Action**



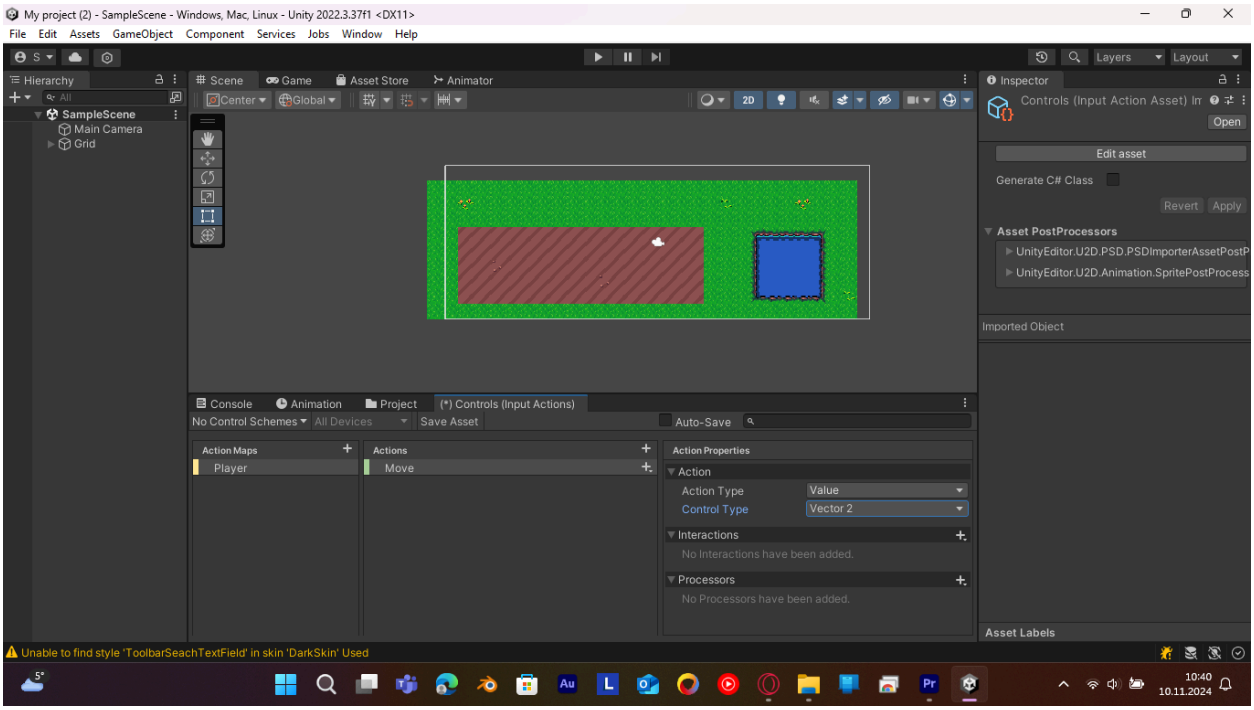
4. Double click on your **Input Action** and open Controls → Rename the Action Map to **Player**



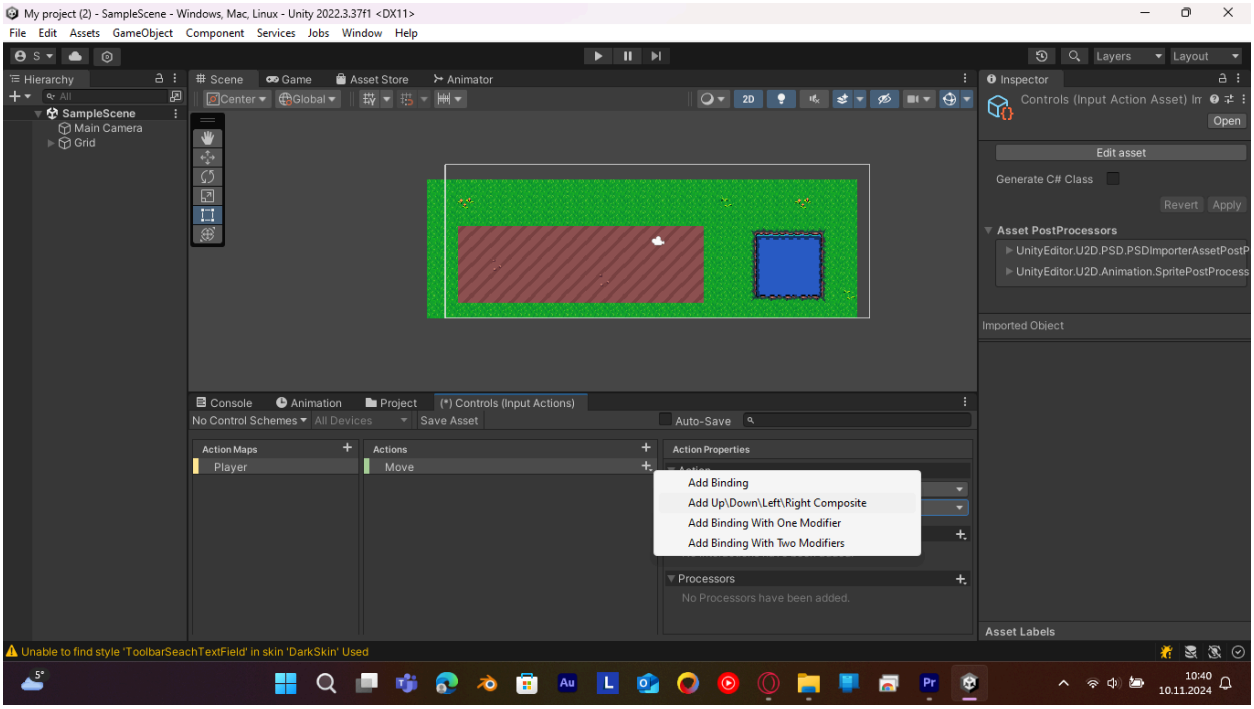
5. Cut or Delete the Binding



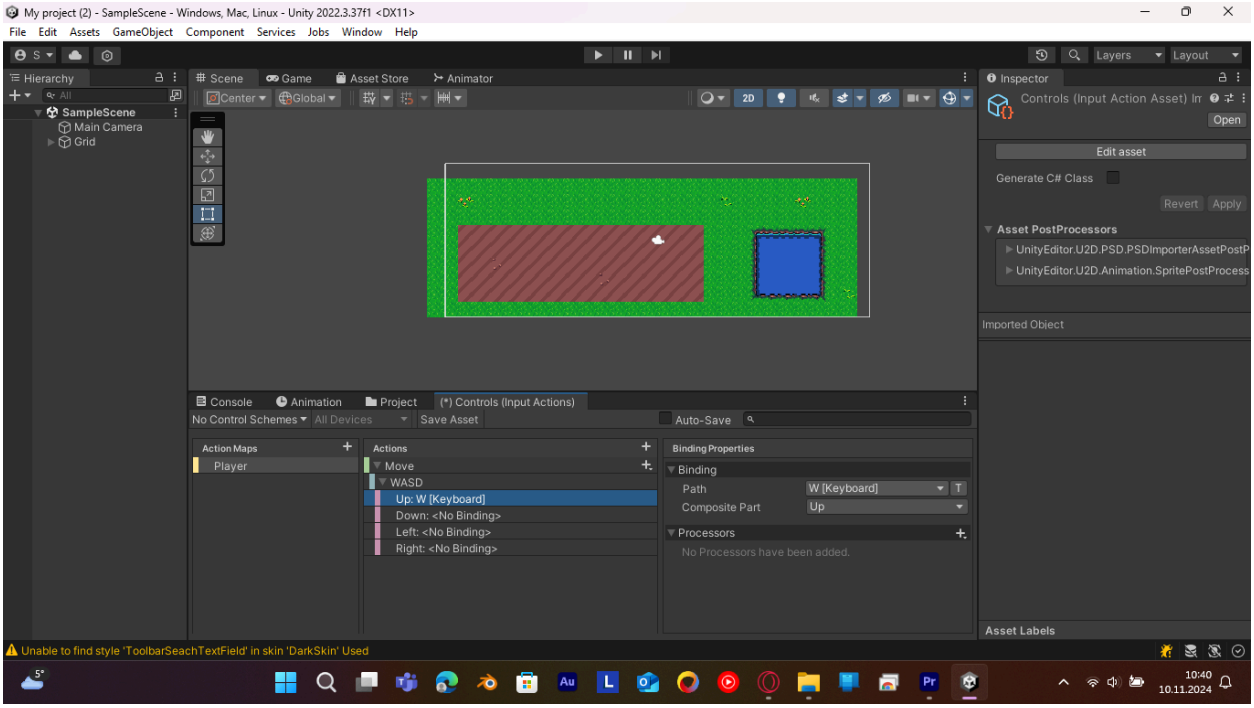
6. Rename the Actions to **Move** and in here change the Action Type to **Value** and the Control Type to **Vector 2**



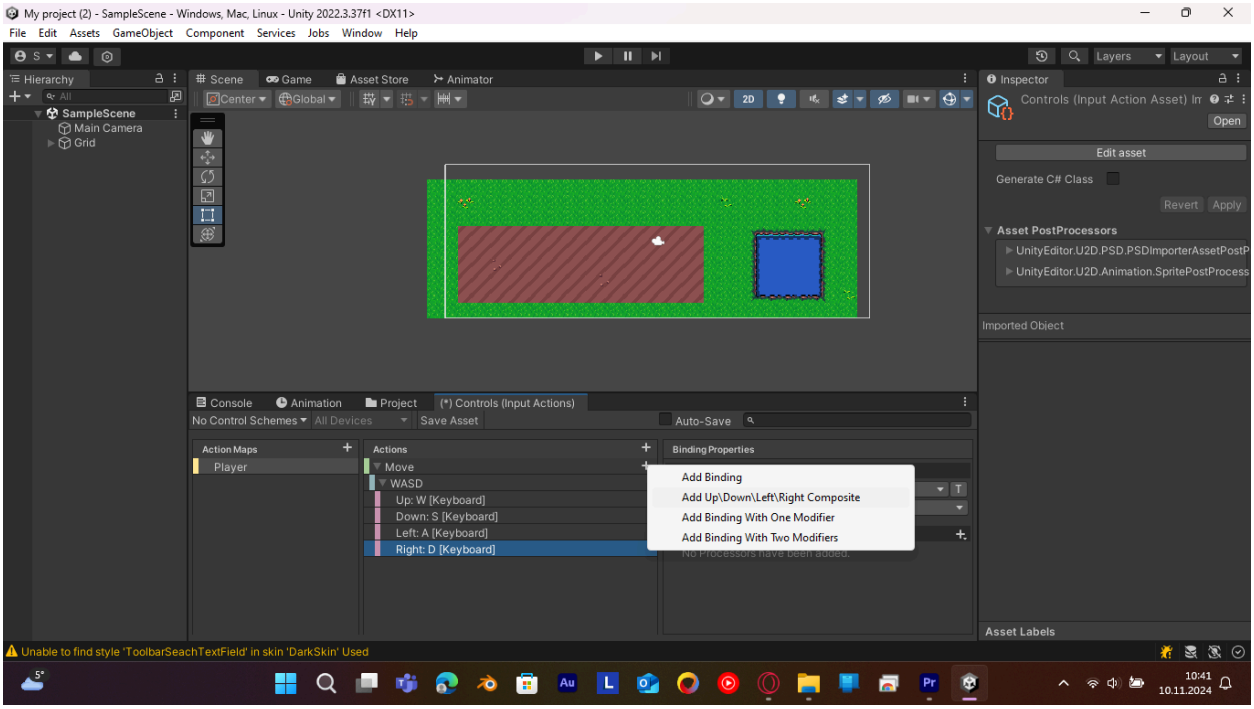
7. Add **Up/Down/Right/Left** Composite to Move and call it **WASD**



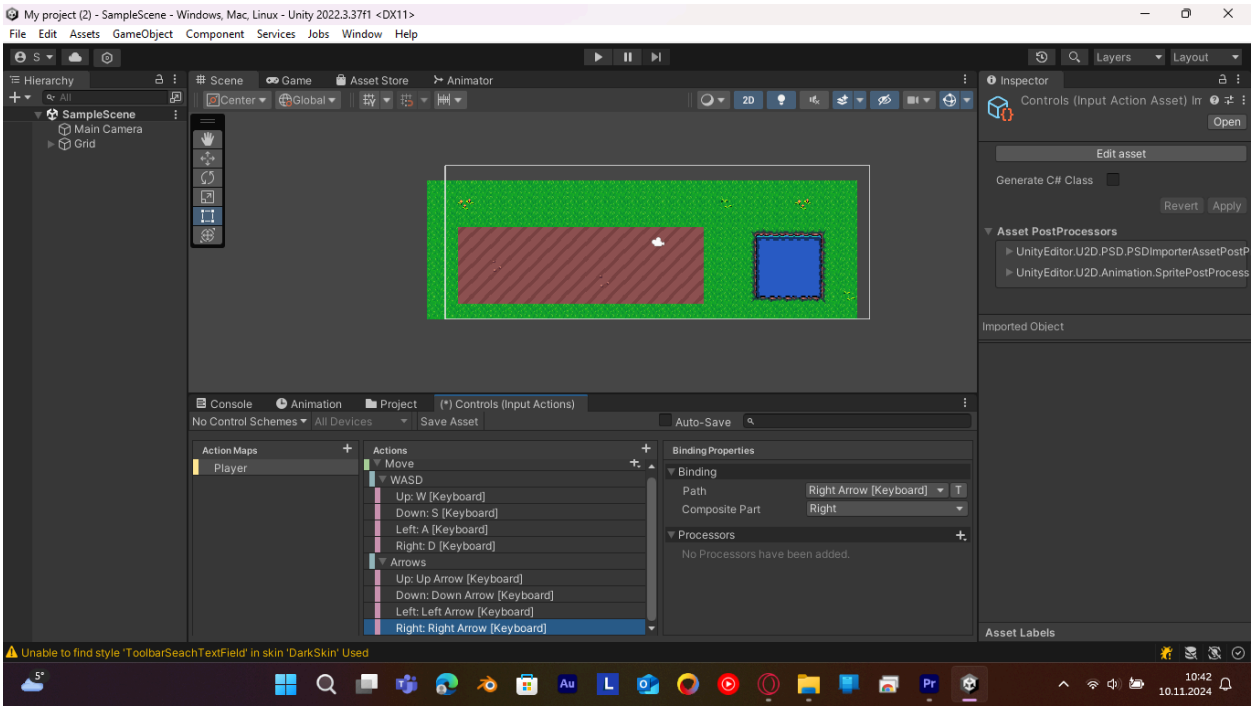
8. Create the right directions for Move (e.g. Up = W, Down = S)



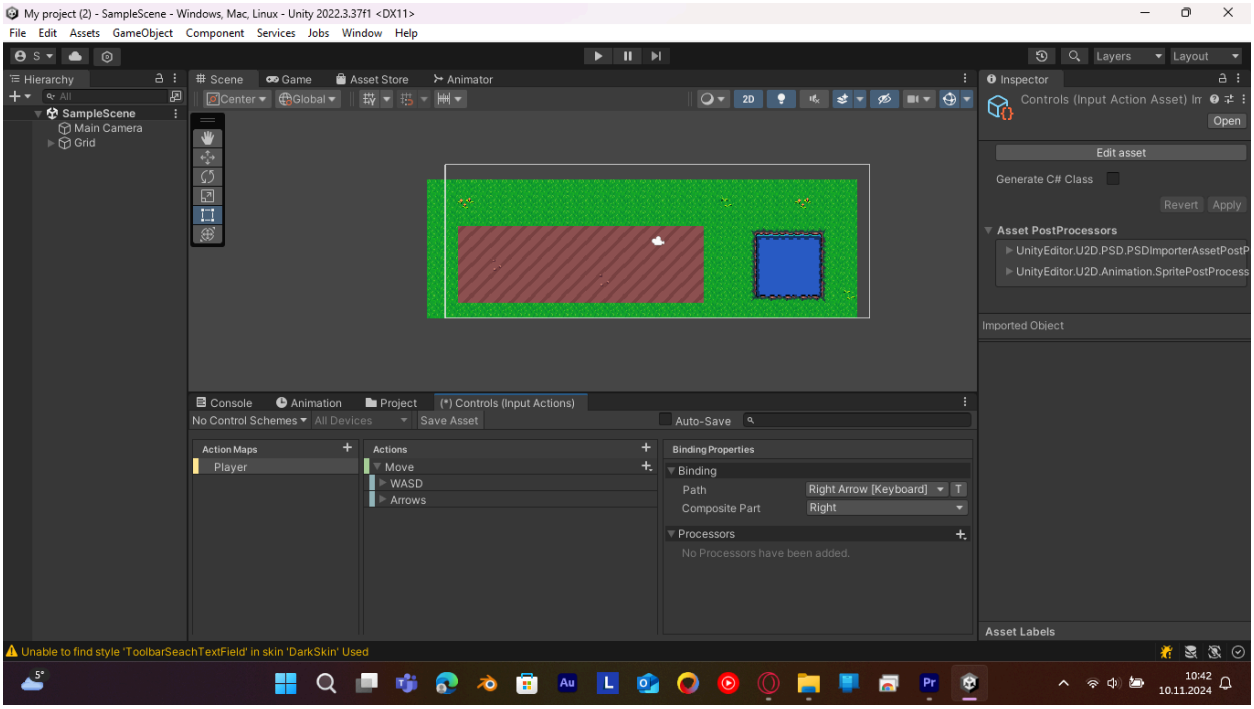
9. Create another **Up/Down/Left/Right** Composite and name it **Arrows**



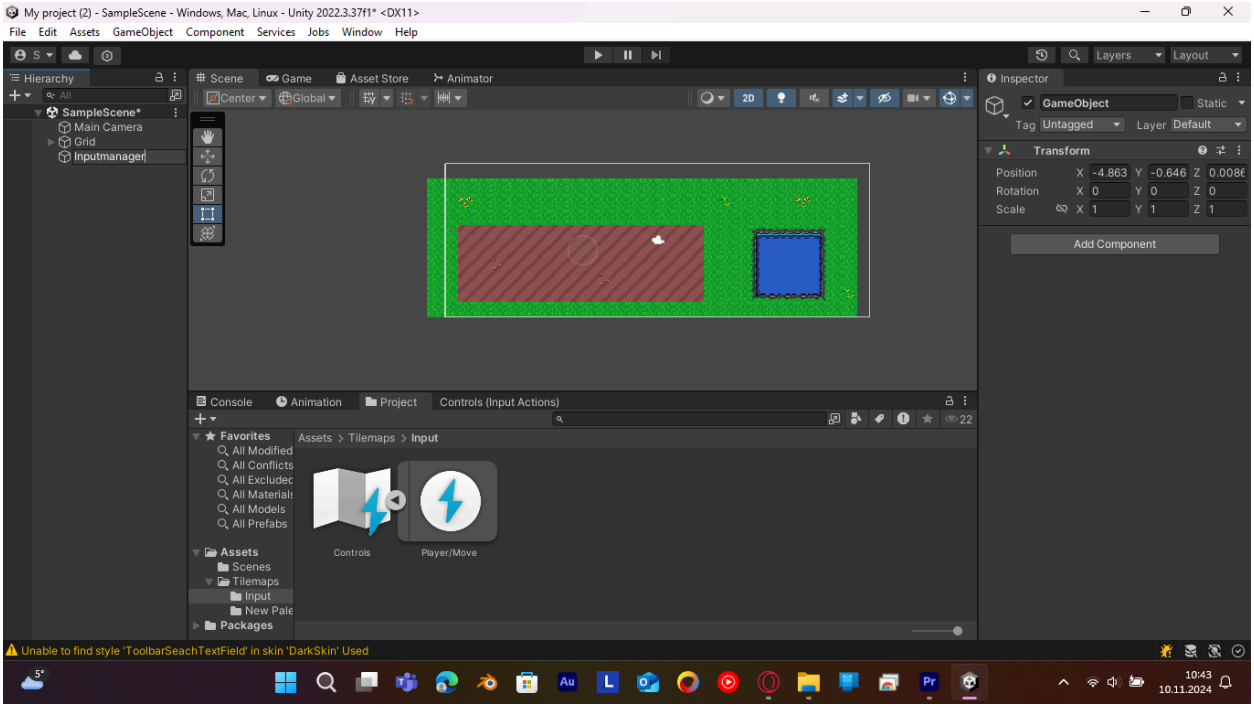
10. Do the same thing for Arrow movements (e.g. Right = Right Arrow, Left = Left Arrow)



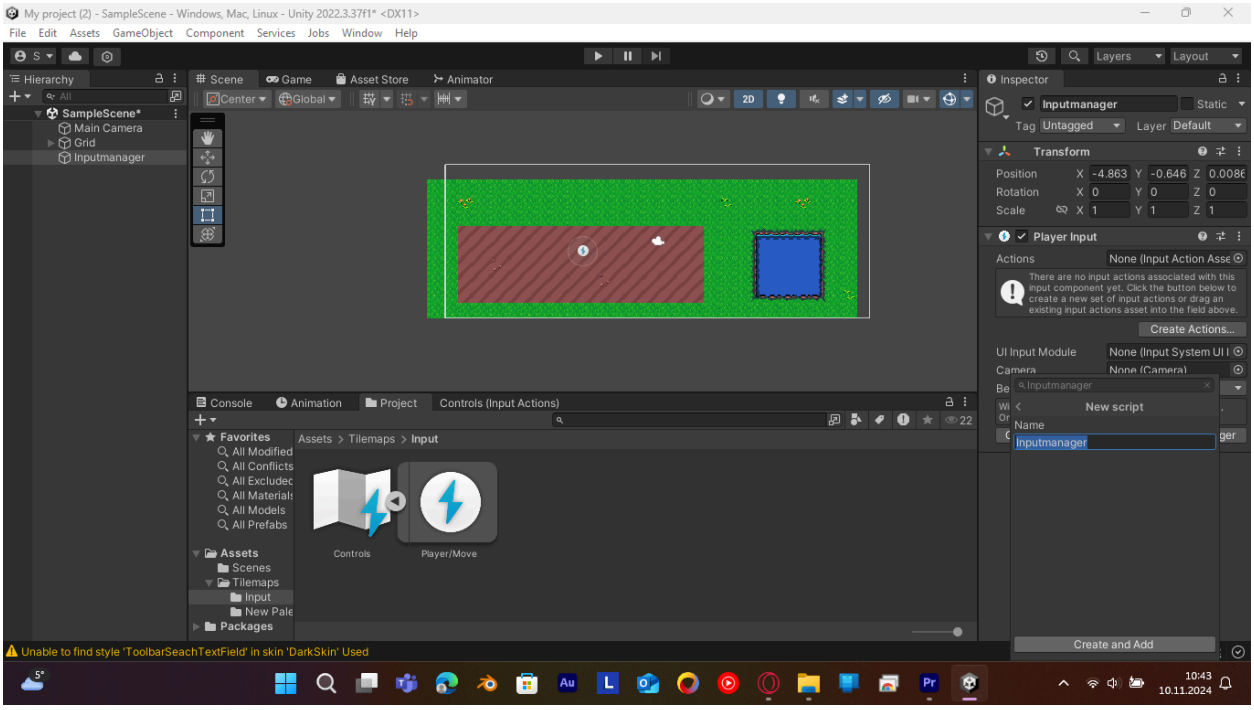
11. Never forget to click on **Save** and save your asset



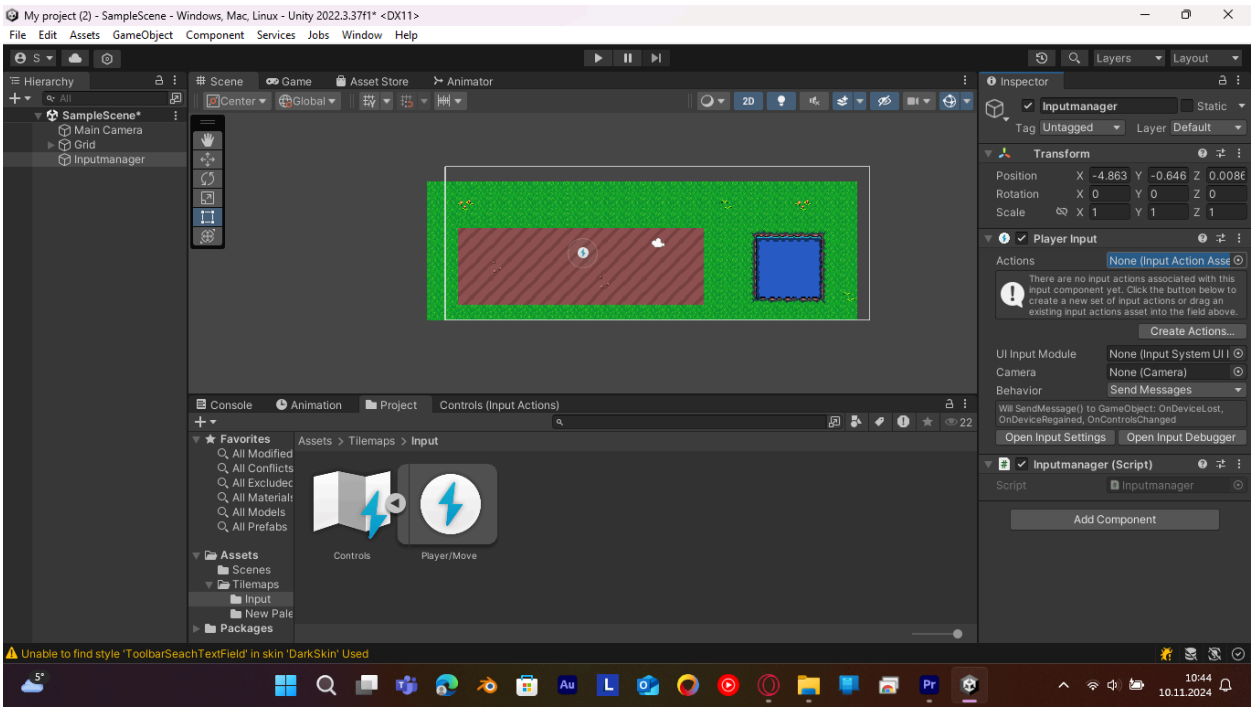
12. Create a new Game Object under Hierarchy and call it Inputmanager



13. Click on Inputmanager and add **Component** → **Player Input** and **new Script** (name it Inputmanager)



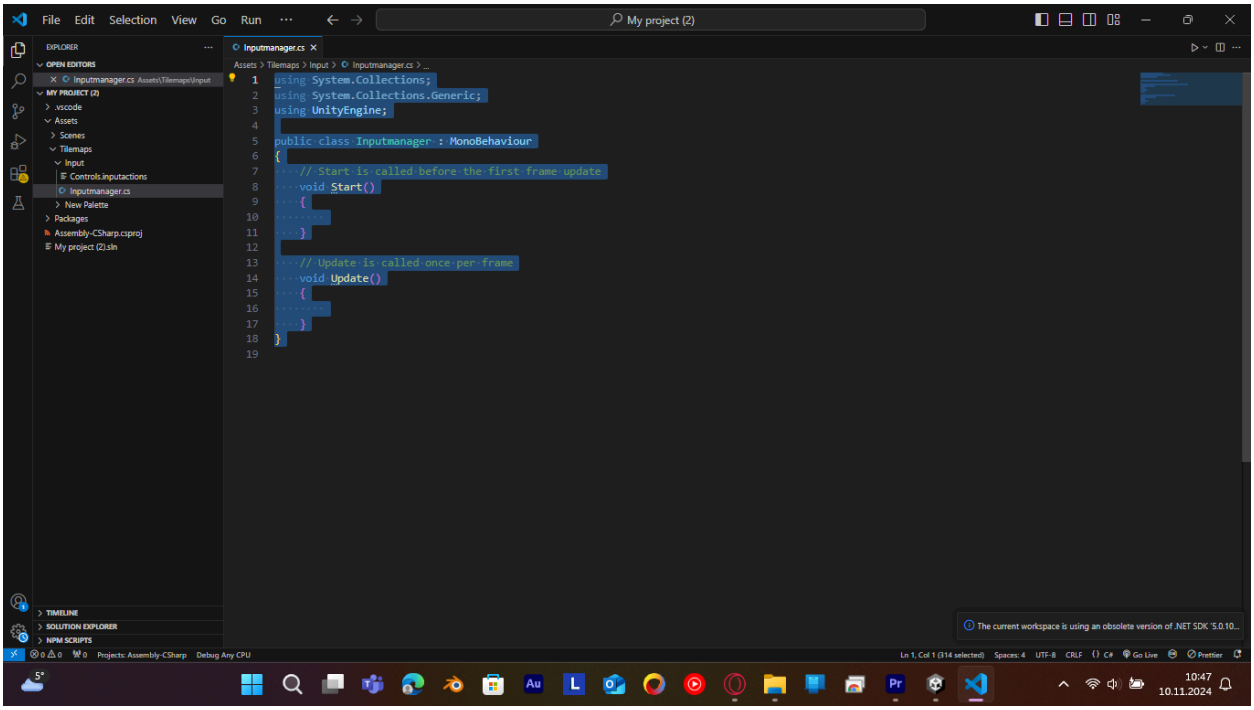
14. Drag and drop your C**ontrols** to the **Actions part** in your **Player Input** and open your script by double clicking on it



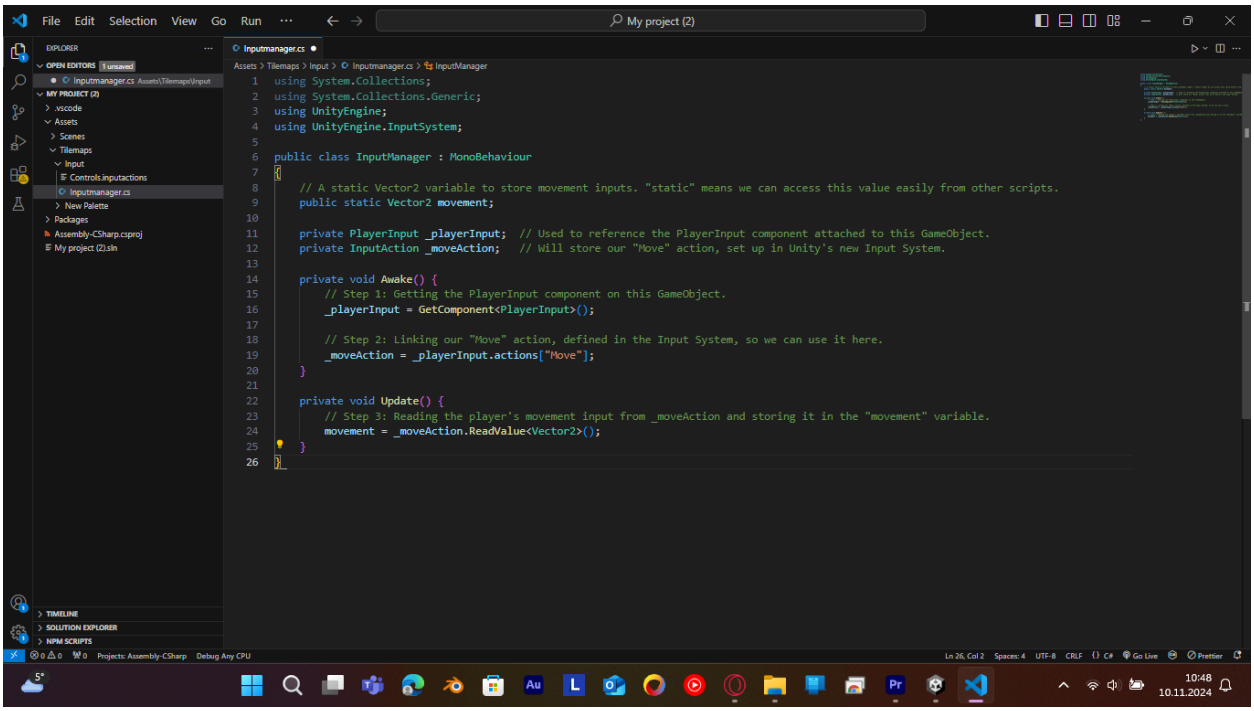
15. Use this Script for your **Inputmanager**

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

public class InputManager : MonoBehaviour
{
    // A static Vector2 variable to store movement inputs. "static" means we can a
    public static Vector2 movement;

    private PlayerInput _playerInput;  // Used to reference the PlayerInput compon
    private InputAction _moveAction;   // Will store our "Move" action, set up in Un

    private void Awake() {
        // Step 1: Getting the PlayerInput component on this GameObject.
        _playerInput = GetComponent<PlayerInput>();

        // Step 2: Linking our "Move" action, defined in the Input System, so we can
```

```
        _moveAction = _playerInput.actions["Move"];
    }

    private void Update() {
        // Step 3: Reading the player's movement input from _moveAction and storir
        movement = _moveAction.ReadValue<Vector2>();
    }
}
```



16. Delete the already **existing Code** and paste the Code from this **Tutorial**



17. Paste the **Code**

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

▼ Explanation

These lines are adding extra features from Unity that we need for this code to work. "**UnityEngine.InputSystem**" is especially important because it allows us to use Unity's new Input System to **manage** player controls.

```
public class InputManager : MonoBehaviour
```

▼ Explanation

This starts a new script called `InputManager`. By putting "MonoBehaviour" here, we tell Unity that this script will be attached to a GameObject

(like our player or camera) and that it will run in Unity's environment.

```
public static Vector2 movement;
```

▼ Explanation

This creates a variable called `movement`. "public" means other scripts can access it, and "static" means it's shared across all copies of `InputManager`. We use it to store the direction the player wants to move in.

```
private PlayerInput _playerInput;
```

▼ Explanation

Here, we create a variable to hold the "PlayerInput" component, which is kinda like a toolkit for managing different controls. We'll use it to connect with our Input System settings.

```
private InputAction _moveAction;
```

▼ Explanation

This variable will store our movement action, specifically what we've set up in the Input System to detect when the player wants to move.

```
private void Awake() {
```

▼ Explanation

This is a special function in Unity. "Awake" runs when our game first starts, and it's used here to set up things before gameplay begins.

```
_playerInput = GetComponent<PlayerInput>();
```

▼ Explanation

Now, we look inside `_playerInput` to find the "Move" action, which we set up in Unity's Input System. This connects our `Move` action with `_moveAction` so we can track when the player is pressing the movement keys.

```
_moveAction = _playerInput.actions["Move"];
```

▼ Explanation

Now, we look inside `_playerInput` to find the "Move" action, which we set up in Unity's Input System. This connects our `Move` action with `_moveAction` so we can track when the player is pressing the movement keys.

```
private void Update() {
```

▼ Explanation

The "Update" function runs every frame (many times per second) and is used here to keep checking the player's input during gameplay.

```
movement = _moveAction.ReadValue<Vector2>();
```

▼ Explanation

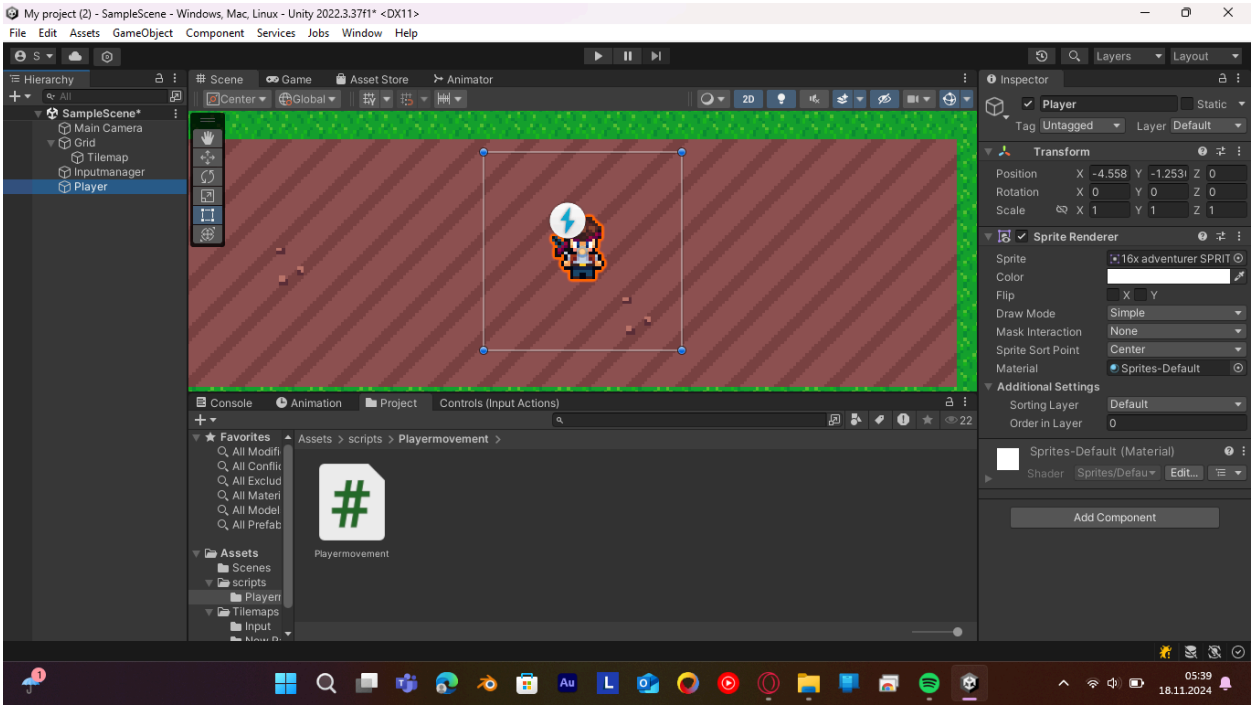
Here, we get the direction the player is moving in (up, down, left, or right) from `_moveAction`. We store that direction in the `movement` variable, which other scripts can access to know where the player wants to go.

2.2 Player Controller script

You've made it this far—well done! Setting up input controls isn't always simple, so if you've followed along, you're doing an amazing job. Each step you complete brings you closer to having full control over your character in-game. By the end of this section, you'll have created a smooth, responsive movement system. This means that after finishing, you'll be able to move your character with ease, setting the foundation for more complex interactions later on. Keep going, and let's bring your game world to life, one step at a time!

Alright, we'll first begin by creating a new a Folder called "Scripts", after that create another Folder called "Playermovement" inside that Folder.

In here we also want to have our Playermovementscript, so create a new c# script and call it "Playermovement" attach this script to our Player.



We now need 2 important features for our Gameobject(Player):

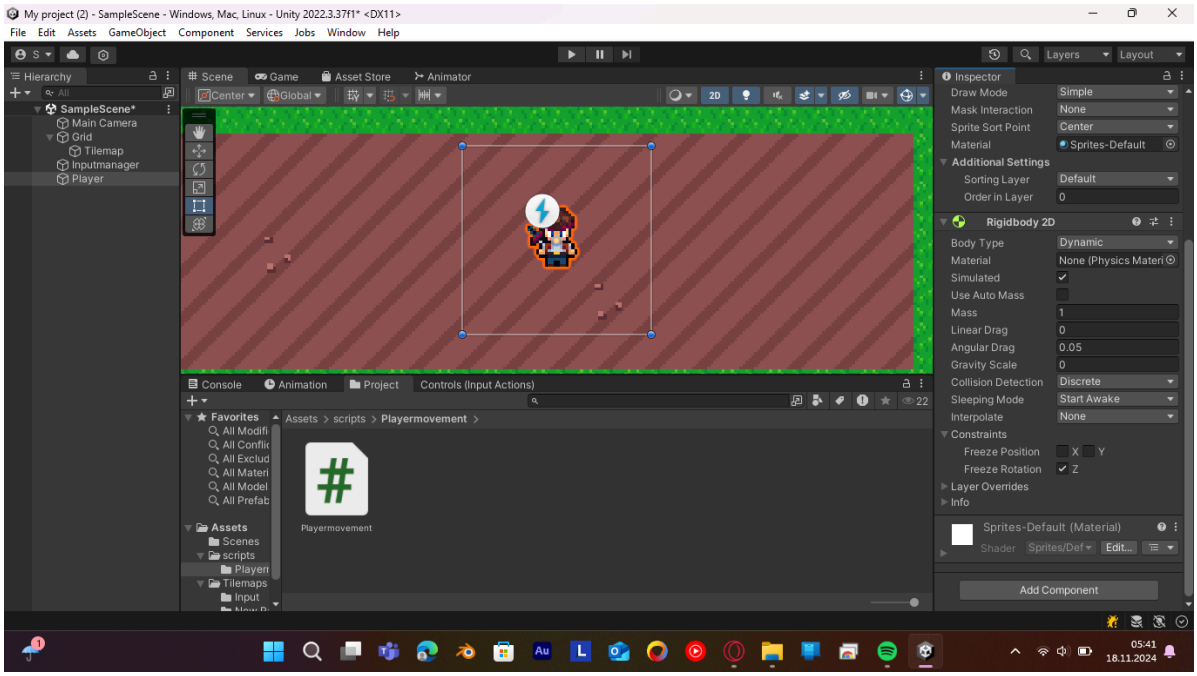
0. ~~Adding the Playermovementscript~~
1. Adding a Rigidbody 2D
2. Adding a Box Collider 2D

Rigidbody 2D:

The **Rigidbody 2D** component handles the physics for our character. By adding it to the character, we're telling Unity to use physics to manage the character's movement, gravity, and collisions. This is essential for 2D movement because it makes the character's interactions feel more realistic and consistent.

Settings for Rigidbody 2D:

- **Body Type**:
  - **Dynamic**: The Rigidbody will be affected by physics forces (like gravity) and can collide with other objects.
- **Gravity Scale**: Controls how much gravity affects the Rigidbody. For top-down games, set this to 0 to prevent the character from falling downward.



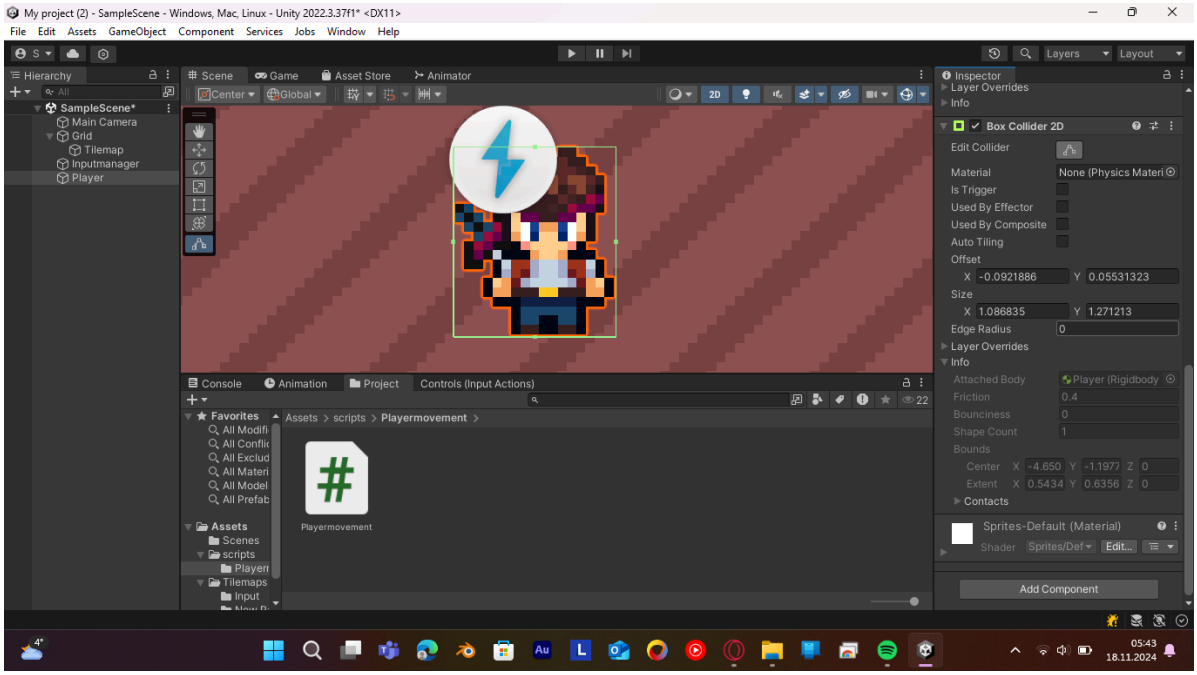
Box Collider 2D:

The **Box Collider 2D** defines the shape of our character in the game world and sets the area where the character can collide with other objects. Imagine it as an invisible box around your character. When the character bumps into walls,

enemies, or other objects, the Box Collider 2D makes sure it responds correctly, either by stopping or bouncing off.

Settings for Box Collider 2D:

- **Size**: Adjusts the height and width of the collider box to fit your character. Make sure it aligns well with your character's shape, so the collisions feel natural.



Now that we are ready, we can start coding!

Open your Playermovement script by double clicking on it and delete the existing code, to bring our character to move we'll need this code.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 5f; // Speed of the player
```

```
    private Rigidbody2D rb; // Reference to the Rigidbody2D for physics-based mo
    private Vector2 movement; // Stores the movement direction

    private void Awake()
    {
        // Get the Rigidbody2D component attached to the player
        rb = GetComponent<Rigidbody2D>();
    }

    private void Update()
    {
        // Read player input directly
        movement.x = Input.GetAxis("Horizontal"); // Input for left/right movement
        movement.y = Input.GetAxis("Vertical");   // Input for up/down movement
    }

    private void FixedUpdate()
    {
        // Apply movement to the Rigidbody2D
        rb.velocity = movement * moveSpeed;
    }
}
```

2.3 Animation Setup

...the full tutorial continues in the Premium Version!

Questions Form