Input

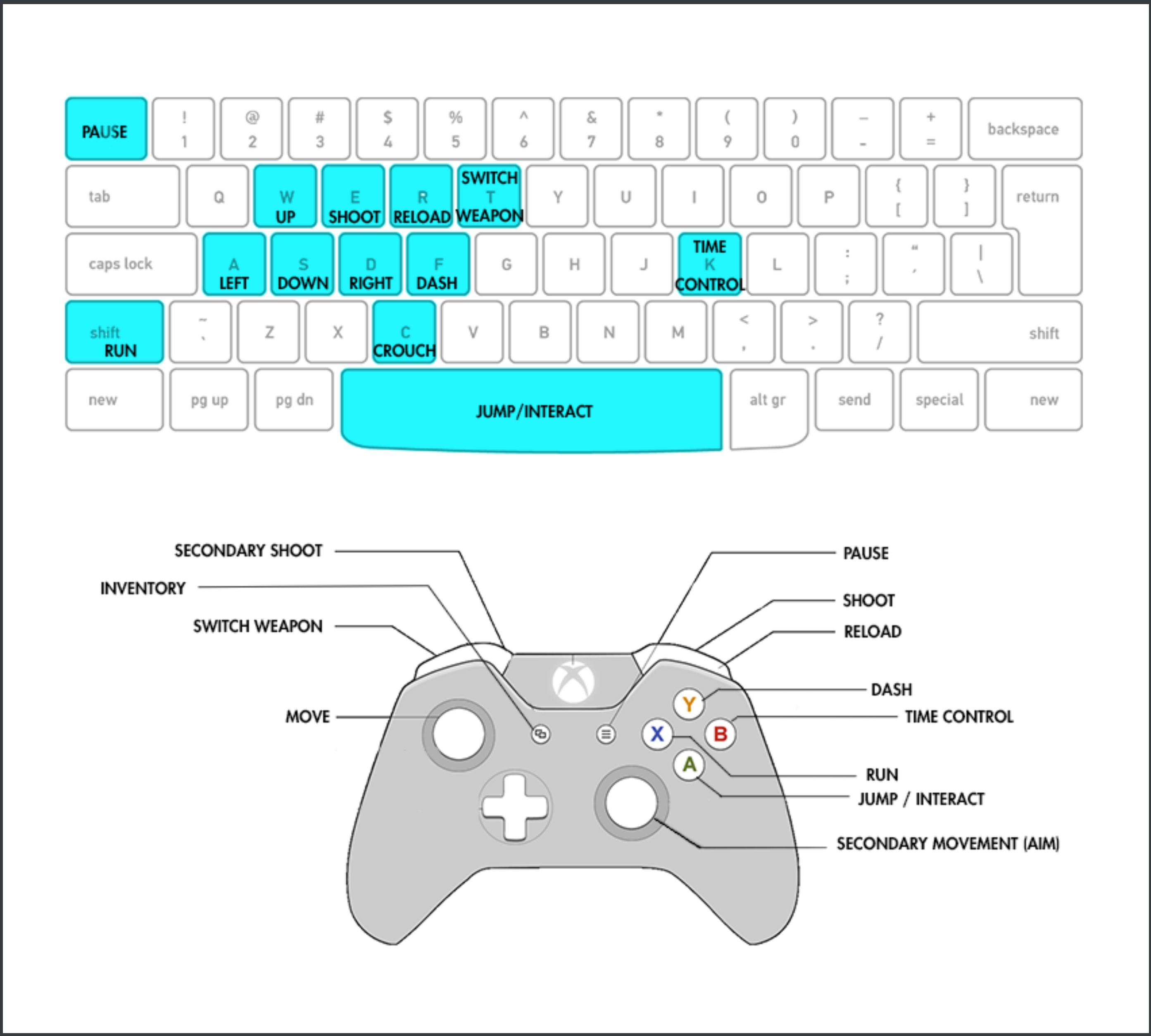
This page explains how to setup input in your game.

Introduction

Input is at the heart of every game. The TopDown Engine is no exception. It currently supports mobile controls (iOS, Android...), keyboard, gamepad (setup for windows xbox pad but feel free to rebind the keys for other controllers), and mouse.

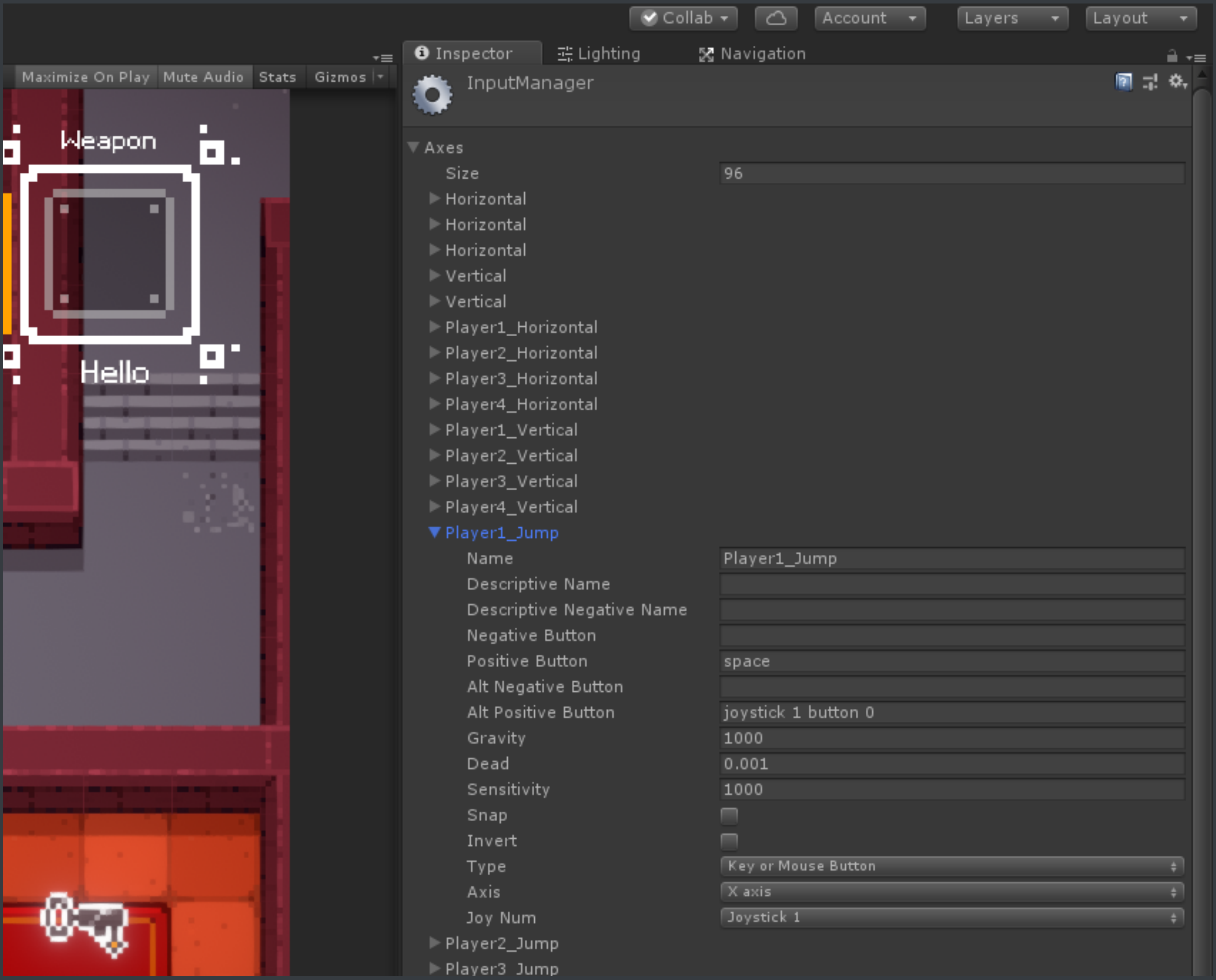
Default Input

By default the keyboard layout has been created to support 4 simultaneous players. Now that may not be very comfortable for a single player game, feel free to remap the buttons to suit your needs (see the following section). In any case, here is the default layout (click on the image to expand) for Player 1 (check out the input settings for other player bindings):



The default keyboard and xbox pad for PC layout for Player 1

Input Settings – How to change key bindings?



InputManager

Axes
Size 96
Horizontal
Horizontal
Horizontal
Vertical
Vertical
Player1_Horizontal
Player2_Horizontal
Player3_Horizontal
Player4_Horizontal
Player1_Vertical
Player2_Vertical
Player3_Vertical
Player4_Vertical
Player1_Jump

Name	Player1_Jump
Descriptive Name	
Descriptive Negative Name	
Negative Button	
Positive Button	space
Alt Negative Button	
Alt Positive Button	joystick 1 button 0
Gravity	1000
Dead	0.001
Sensitivity	1000
Snap	☐
Invert	☐
Type	Key or Mouse Button
Axis	X axis
Joy Num	Joystick 1

Player2_Jump
Player3_Jump

That's a lot of axis

For keyboard and gamepad (and to a lesser extent mouse), key bindings are defined, like in any good Unity project, **via the native Input settings**. You can access these via the top menu Edit > Project Settings > Input. This should open a quite large list of "axis". If you've never seen this panel before, maybe have a look at the official documentation first.

Now that list is pretty long. That list has been filled to support 4 players (you could add more if you want), and proposes one key configuration. Feel free to **edit each of these** to rebind the keys to your preference.

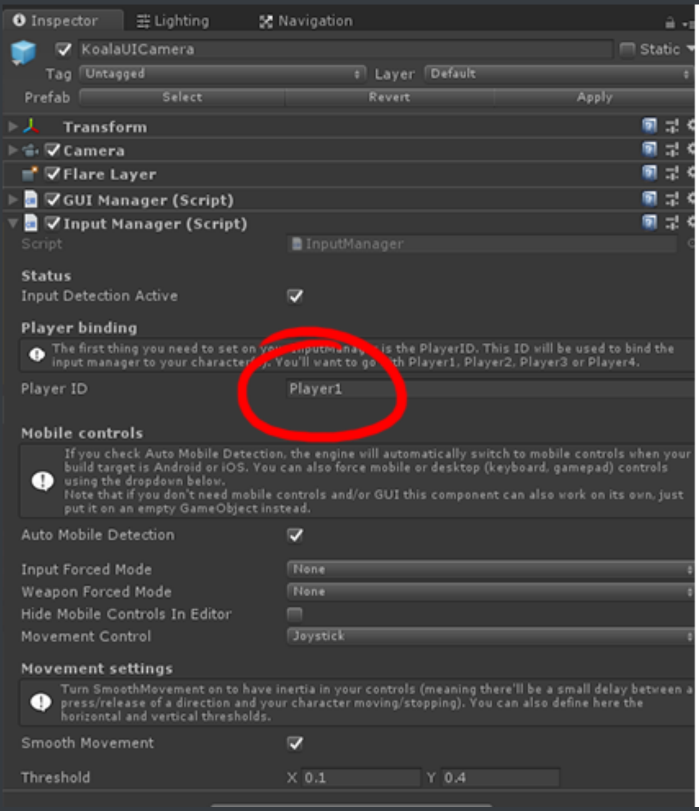
Input Manager and GUI

To move your character on screen, you usually use **input**, whether it's via touch on a screen or by pressing mechanical keys. To handle input detection and pass it to your Character, the TopDown Engine uses an Input Manager. Most levels of the TopDown Engine have a UI Camera in them. This UI Camera prefab also has, nested inside it, a bunch of mobile controls used to control the player when playing on mobile. For that reason, and to make binding easier, you'll find the InputManager at the top level of the UICamera prefab. You can decide to place it on its own game object if you prefer.

The **Input Manager** is quite a long class. If you take a look at it, you'll see it's actually **not too complex**, just very verbose. It has a lot of methods that you'll be able to bind to your mobile controls, and manages a small state machine for each of the registered buttons.

From its inspector you'll be able to specify the **PlayerID** (see the next section for more on that), whether or not you want to use mobile detection or force mobile controls. You can also decide whether you prefer virtual arrows/dpad or virtual joystick. And finally you can set smooth movement on or off (should a press on left be instant or quickly lerped?), and the horizontal and vertical thresholds.

Player ID



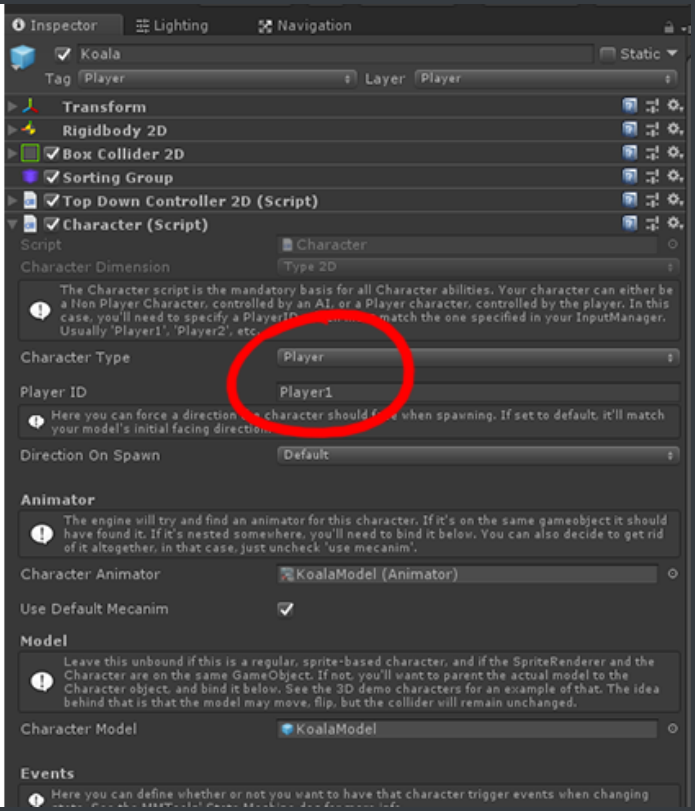
KoalaUICamera
Tag: Untagged
Layer: Default
Transform
Camera
Flare Layer
GUI Manager (Script)
Input Manager (Script)
Script: InputManager

Status
Input Detection Active: ✓

Player binding
The first thing you need to set on [illegible] is the PlayerID. This ID will be used to bind the input manager to your character[illegible]. You'll want to go [illegible]th Player1, Player2, Player3 or Player4.
Player ID: Player1

Mobile controls
If you check Auto Mobile Detection, the engine will automatically switch to mobile controls when your build target is Android or iOS. You can also force mobile or desktop (keyboard, gamepad) controls using the dropdown below.
Note that if you don't need mobile controls and/or GUI this component can also work on its own, just put it on an empty GameObject instead.
Auto Mobile Detection: ✓
Input Forced Mode: None
Weapon Forced Mode: None
Hide Mobile Controls In Editor: ☐
Movement Control: Joystick

Movement settings
Turn SmoothMovement on to have inertia in your controls (meaning there'll be a small delay between a press/release of a direction and your character moving/stopping). You can also define here the horizontal and vertical thresholds.
Smooth Movement: ✓
Threshold: X 0.1 Y 0.4



Koala
Tag: Player
Layer: Player
Transform
Rigidbody 2D
Box Collider 2D
Sorting Group
Top Down Controller 2D (Script)
Character (Script)
Script: Character
Character Dimension: Type 2D
The Character script is the mandatory basis for all Character abilities. Your character can either be a Non Player Character, controlled by an AI, or a Player character, controlled by the player. In this case, you'll need to specify a PlayerID [illegible] match the one specified in your InputManager. Usually 'Player1', 'Player2', etc.
Character Type: Player
Player ID: Player1
Here you can force a direction [illegible] character should f[illegible]e when spawning. If set to default, it'll match your model's initial facing direction.
Direction On Spawn: Default

Animator
The engine will try and find an animator for this character. If it's on the same gameobject it should have found it. If it's nested somewhere, you'll need to bind it below. You can also decide to get rid of it altogether, in that case, just uncheck 'use mecanim'.
Character Animator: KoalaModel (Animator)
Use Default Mecanim: ✓

Model
Leave this unbound if this is a regular, sprite-based character, and if the SpriteRenderer and the Character are on the same GameObject. If not, you'll want to parent the actual model to the Character object, and bind it below. See the 3D demo characters for an example of that. The idea behind that is that the model may move, flip, but the collider will remain unchanged.
Character Model: KoalaModel

Events
Here you can define whether or not you want to have that character trigger events when changing

Your InputManager's PlayerID (right) must match the one on the Character(s) you want to control

One thing here that is **really important** to understand is the **PlayerID** notion. The engine is able to send input from the InputManager to playable characters in the scene (objects with a Character component whose type is set to Player). Each InputManager has a PlayerID attribute, and it'll send its information **to all characters whose own PlayerID attribute match it**. So there are a few ways this can work :

- The most common way is to simply **let the engine handle this**. Add an InputManager to your scene, set its PlayerID to "Player1". Then in the LevelManager, when you add your character, have the "**Auto attribute player IDs**" checkbox checked, and it'll automatically give the playable character the "Player1" ID. If you are in a multiplayer level use "Player2", "Player3" and "Player4". You can add your own of course, but these are the default ones.
- You can also set the PlayerID **on the prefab**. For example you could have a Dog character, whose PlayerID is "JoeTheDog" (why not?). If you add an InputManager to your scene and want it to target that Dog character, set its own PlayerID parameter to "JoeTheDog". Press play and you should be able to control your Dog character. That method is great if for some reason you don't want to use the Level Manager and want to instantly move a prefab in the scene.

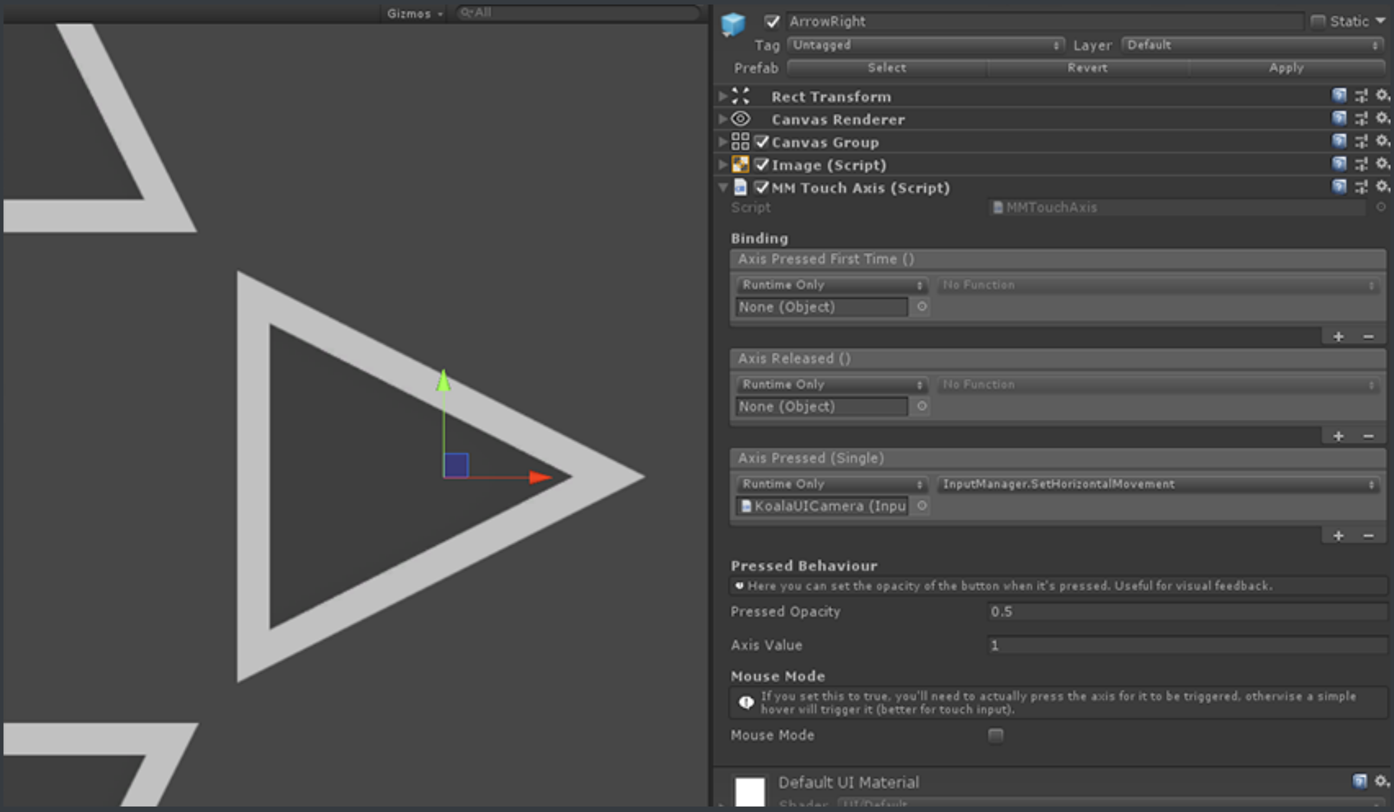
I use that second method a lot **when tweaking a character**. Let's say I want to change the Jump behaviour of my character. I'll just copy my character's prefab, put that copy in the scene, set its PlayerID to Player1. Then in the LevelManager I put the original prefab as the Player Prefab. I press play and I'm now controlling two characters with the same input. I can then change the jump settings on one of them (or even add a new Jump script in place of the old one), and **play test both very fast** to see if my changes are improving the gameplay.

How does platform detection work?

From the InputManager's inspector you can turn **auto mobile detection** on or off.

If you unfold the UICamera prefab, you'll see it contains quite a lot of stuff, and notably Arrows, Joystick and Buttons canvas groups. They're **disabled by default** to allow for simpler level editions but you can turn them on if you prefer, or to tweak them. These are then bound to the GUIManager component (among other stuff). Mobile detection works in a very simple way : if you're targeting a mobile platform (iOS or Android), it'll show the controls when you press play. If you're targeting another platform, it'll **hide and disable them**. You can also force one mode or the other from the inspector.

Virtual Arrows, Joysticks and Buttons



ArrowRight

Tag: Untagged
Layer: Default

Script: MMTouchAxis

Binding

Axis Pressed First Time ()
Runtime Only | No Function
None (Object)

Axis Released ()
Runtime Only | No Function
None (Object)

Axis Pressed (Single)
Runtime Only | InputManager.SetHorizontalMovement
KoalaUICamera (Inpu

Pressed Behaviour

Here you can set the opacity of the button when it's pressed. Useful for visual feedback.

Pressed Opacity: 0.5

Axis Value: 1

Mouse Mode

If you set this to true, you'll need to actually press the axis for it to be triggered, otherwise a simple hover will trigger it (better for touch input).

Mouse Mode: ☐

Shader: UI/Default

The Touch Axis (arrows) inspector

The asset comes packed with a fully functional arrow D-pad, a joystick and some buttons, but of course **you can remix all that**, add buttons, add joysticks, etc. to match your own gameplay. You'll find a demo of all the ready-to-use mobile input building blocks included in the engine in the **MMControlsTestScene** demo scene.

Mobile arrows are very simple, you just need an MMTouchAxis component on a Rect+CanvasGroup object. Then in its inspector you need to set the axis value (usually -1 for left/down or 1 for right/up). Then you need to bind it to your InputManager, simply on the AxisPressed event. To do that, just drag your InputManager (in most cases the UICamera prefab in your scene) onto the little box below "runtime only", and then select the appropriate method (SetVerticalMovement, SetHorizontalMovement, or their secondary counterparts).

Mobile joysticks are even easier to setup. Just add a MMTouchJoystick component to a Rect+CanvasGroup object, define which axis are enabled (you may want an horizontal only stick for example), the max range (how far the knob can move from its base), and then you bind your input manager to it. Note that you'll also need to specify a target camera (usually your UICamera).

Repositionable joysticks : similar in aspect to the regular joysticks, these ones reposition within a bigger zone as you lift your thumb and put it back on the screen again.

Buttons work in the same way too, but for them you'll be able to specify three different events : button down (when it's pressed for the first time), button pressed (every frame as long as this button remains pressed), and button up (when it's released). This will allow you in your Character Abilities (for example) to call certain methods when a button gets released for example. You don't have to bind all events if you're not using all of them.

Swipe zones : let you drag your thumb across their surface, and trigger swipe events, complete with angle, direction, length, etc.

Customizing mobile input

Using the classes above, you should be able to customize your mobile controls any way you want. For example, if you wanted to add a second joystick to the Loft3D demo scene, and have it control the shotgun that can get picked in that scene, you'd go through the following simple steps :

- select the **ShotgunWithAmmo** prefab in your project view, set its aim control to PrimaryThenSecondaryMovement
- select your **UICamera**, set **InputForcedMode** to Mobile (so that you can test it in editor)
- duplicate the **Joystick** node under the UICamera's canvas, move it a bit to the right
- on its joystick knob, on the **MMTouchJoystick** component, change the Binding to InputManager.SetSecondaryMovement (the **vector2** version)

- you're done, press **play** and have fun!

Disabling input

If you wish to disable input, you can do it in two main ways :

- At the InputManager level, by setting InputDetectionActive to false (either via script or via its inspector). This will prevent all input from being read via the InputManager.
- At the ability level, by setting AbilityPermitted to false

Nice Touch

The TopDown Engine includes another of More Mountains' assets : Nice Touch. **Don't buy it if you already own the TopDown Engine!** I created Nice Touch to provide **a simple and fast input solution**. It handles keyboard, gamepad, mouse, and touch input. There are a lot of other input solutions out there. Nice Touch/MMControls were created to give a simpler alternative, faster to setup, without unnecessary settings. Feel free to not use these and replace them with your own.

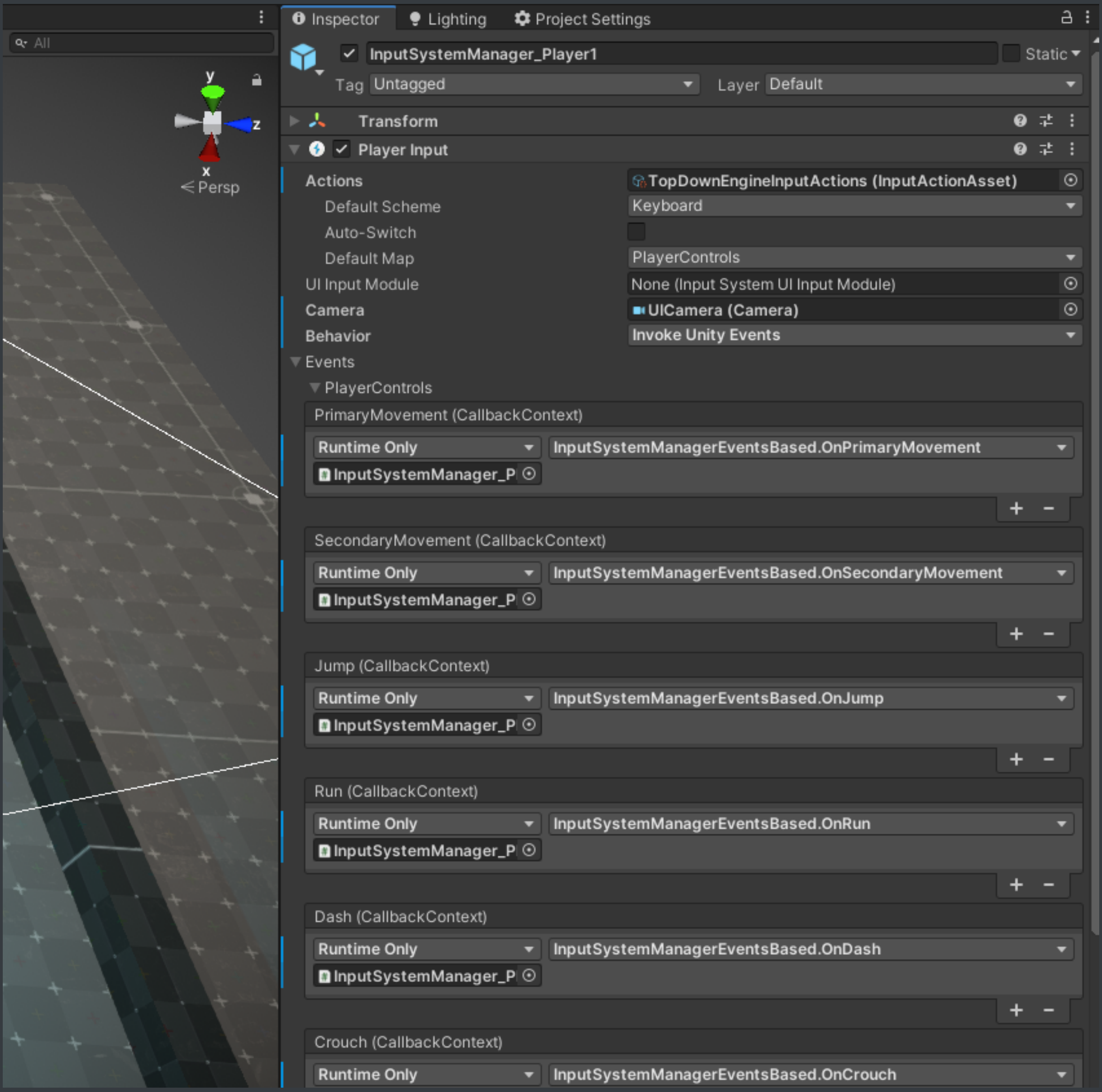
You'll find these scripts into MMTools/MMControls. That same folder also includes a test scene : MMControlsTestScene. You'll want to keep that folder in your game.

Using Unity's "new" input system

Unity released a **new input system** 3 years ago now, and despite still not being fully documented or stable, it's been marked as "verified" recently. If despite all the **risks** associated to using it you want to use it in your production, the TopDown Engine comes with two examples of it in use. You'll find them under TopDownEngine/Common/ScriptsInputSystem, and of course they require that you **install and setup** the Input System package.

You'll notice there that the InputManager gets replaced by an **InputSystemManager**, or **InputSystemManagerEventsBased**, depending on what demo you look at (the event version is best suited for multiplayer). In both cases you'll find the InputSystemManager under the camera rig. Both rely on another class, TopDownEngineInputActions, which is just the regular auto generated output of the "new" system. Aside from the specific API calls of that system, and the use of that class, there's nothing specific to it, it works like the regular version. It's recommended that you have a good understanding of that system to pick this option, like the rest of that system, it's less intuitive than the classic option.

After you've installed the InputSystem package, it's possible that Unity will lose some references, and you may have to setup bindings manually. This should only happen in the MinimalScene3D_InputSystem_Multiplayer scene, the other one should be safe. If that happens, this is what your InputSystemManager_Player1 and InputSystemManager_Player2's inspectors should look like :



InputSystemManager_Player1
Tag Untagged Layer Default
Transform
Player Input

Actions	TopDownEngineInputActions (InputActionAsset)
Default Scheme	Keyboard
Auto-Switch	
Default Map	PlayerControls
UI Input Module	None (Input System UI Input Module)
Camera	UICamera (Camera)
Behavior	Invoke Unity Events

Events
PlayerControls

PrimaryMovement (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnPrimaryMovement
InputSystemManager_P

SecondaryMovement (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnSecondaryMovement
InputSystemManager_P

Jump (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnJump
InputSystemManager_P

Run (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnRun
InputSystemManager_P

Dash (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnDash
InputSystemManager_P

Crouch (CallbackContext)
Runtime Only | InputSystemManagerEventsBased.OnCrouch

The InputSystemManager_Player1 inspector in the MinimalScene3D_InputSystem_Multiplayer scene