KINERACTIVE 1.11

Component Reference

So much stuff!

This component reference document lists just about everything (I think!) so if you find it difficult to navigate, or that something is not documented well enough, please let me know on the support email contact CleanShirtLabs@gmail.com and I will gladly help you get the most out of KINERACTIVE.

Managers

Kineractive Manager

The **Kineractive Manager** is the main component of KINERACTIVE. Without it in the scene, nothing much will happen.

The main function of the **Kineractive Manager** is to fire a raycast at regular intervals, and to a set distance. When a raycast hits a collider, the **Kineractive Manager** checks to see if that collider also has an **Input Handler** attached. If an **Input Handler** detected, then it will enable this Input **Handler** component, which now allows the player to make an interaction of some sort.

The second function of the **Kineractive Manager**, is that it serves as a hub for all important fields, various scripts can find, check and change UI, Audio, and IK related Transforms in one central area. For the programmers out there: it is a singleton.

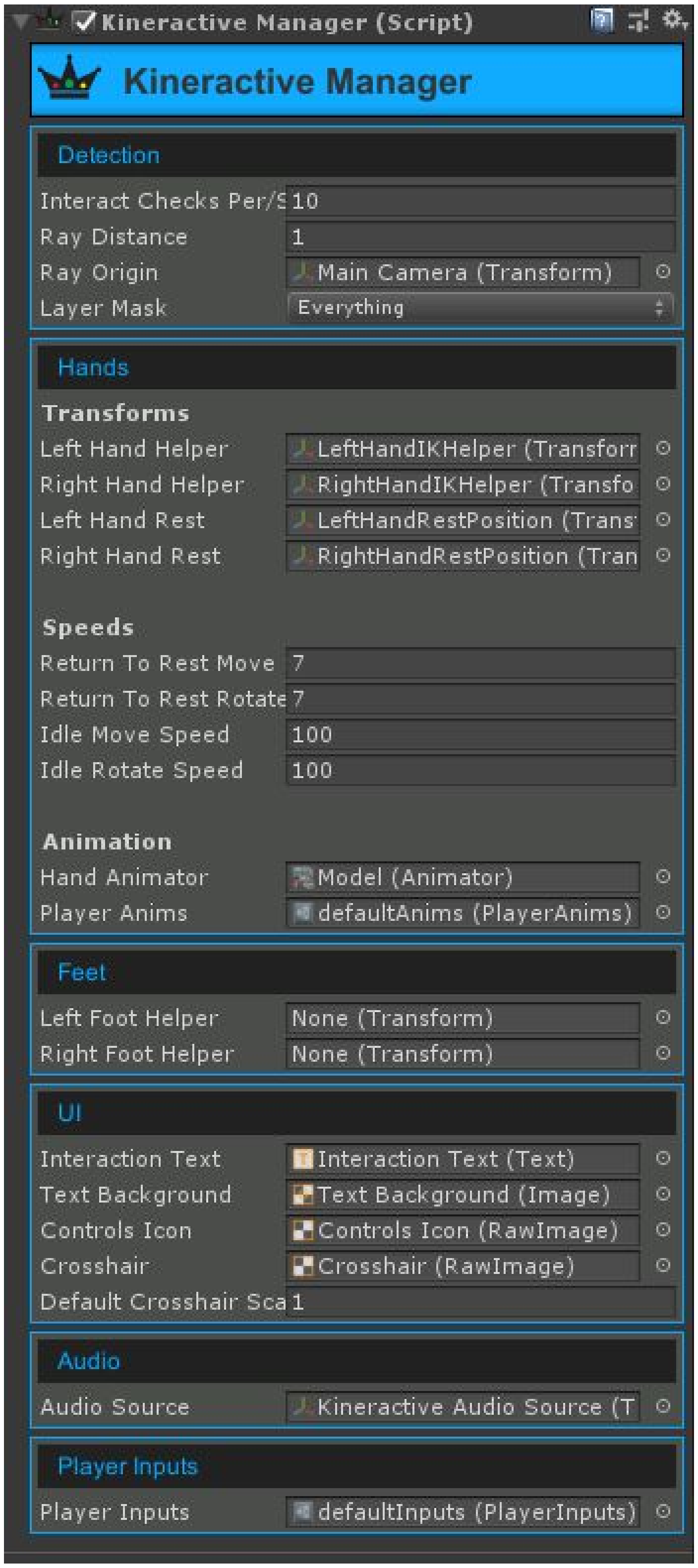
TIP: *It's good to have the **Kineractive Manager** in an easily accessible area of the Hierarchy until you have finished filling out all of the fields, after that you can tuck it away somewhere if you like to keep your scene tidy and organised.*

Add it to the scene

To add a **Kineractive Manager** component to your project:

- Go into the Project window and look in the Assets \ KINERACTIVE \ Scripts \ Prerequisite folder and drag and drop the **Kineractive Manager** component into your selected GameObject (either in the Hierarchy or the Inspector).

- Or select your GameObject, and in the Inspector click the *Add Component* button, then in the popup menu select KINERACTIVE > Prerequisite



Kineractive Manager (Script)

Kineractive Manager

Detection

Interact Checks Per/S	10
Ray Distance	1
Ray Origin	Main Camera (Transform)
Layer Mask	Everything

Hands

Transforms

Left Hand Helper	LeftHandIKHelper (Transforr
Right Hand Helper	RightHandIKHelper (Transfo
Left Hand Rest	LeftHandRestPosition (Trans
Right Hand Rest	RightHandRestPosition (Tran

Speeds

Return To Rest Move	7
Return To Rest Rotate	7
Idle Move Speed	100
Idle Rotate Speed	100

Animation

Hand Animator	Model (Animator)
Player Anims	defaultAnims (PlayerAnims)

Feet

Left Foot Helper	None (Transform)
Right Foot Helper	None (Transform)

UI

Interaction Text	Interaction Text (Text)
Text Background	Text Background (Image)
Controls Icon	Controls Icon (RawImage)
Crosshair	Crosshair (RawImage)
Default Crosshair Sca	1

Audio

Audio Source	Kineractive Audio Source (T

Player Inputs

Player Inputs	defaultInputs (PlayerInputs)

Detection

Interact Checks Per/S

Necessity: *Required*

Accessibility: *Public Read Only Property*

This field takes in a float which determines how many times the raycast is fire per second. I find 10 per second is responsive enough, but perhaps you might want to make it up to 60 raycasts per second if you need an ultra responsive feel in your project. And of course, you can lower the amount of raycasts per second it if you are trying to squeeze every last bit of performance out of your game.

Ray Distance

Necessity: *Required*

Accessibility: *Public Read Only Property*

This float value determines how far (in Unity units) the raycast will reach. This will depend specifically on what your project needs. It's a master setting for the maximum allowed detection distance. (*You can set interaction distance on each interactive object individually (thru the Input Handler on each object) so this should match the or exceed the longest distance on any of your interactive items/objects*) If you just want your character to interact with things at arm's length, then make this value just longer than an arm's length. But if you also want to interact with things far away (e.g. you might have a telekinesis ability) then you can increase the ray's distance to a suitable length.

Ray Origin

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is where the raycast shoots out from. In most cases the camera is the best spot for this, that way wherever your crosshair looks, the ray shoots in the same spot, along the same path.

But perhaps you might have a cursor based game, like a MOBA or ARPG. And then it might be more suitable to have the raycast originate from the cursor's location.

Layer Mask

Necessity: *Required*

Accessibility: *Public Read Only Property*

Use this to filter which objects you want the raycast to interact with. E.g. you may want to filter out the character model, so that the ray shoots through the hands/arms/feet, allowing you to interact with whatever is under them. Or you can put all of your KINERACTIVE objects into their own separate layer, and have the raycast only interact with these specific objects.

Hands

Left Hand Helper

Necessity: *Required*

Accessibility: *Public Read Only Property*

Place a blank Transform GameObject into here, ideally labeled something obvious such as **Left Hand IK Helper**. KINERACTIVE works by moving around an empty Transform (one for each hand). And then the Unity IK system follows these empty Transforms. Therefore, wherever this Transform moves and rotates to, so will the character's hands (via the IK system).

Right Hand Helper

Necessity: *Required*

Accessibility: *Public Read Only Property*

Place a blank Transform GameObject into here, ideally labeled something obvious such as **Right Hand IK Helper**. KINERACTIVE works by moving around an empty Transform (one for each hand). And then the Unity IK system follows these empty Transforms. Therefore, wherever this Transform moves and rotates to, so will the character's hands (via the IK system).

Left Hand Rest

Necessity: *Required*

Accessibility: *Public Read Only Property*

This will be the position/rotation of where the hand is returns to when no interactions are being activated. For example you might want your character's hand to go back to holding the front of a gun, or sword on their hip, or back to a steering wheel, or onto their lap etc.

Parent these *Rest* Transforms to your player or vehicle so that they move along relative to your character.

Right Hand Rest

Necessity: *Required*

Accessibility: *Public Read Only Property*

This will be the position/rotation of where the hand is returns to when no interactions are being activated. For example you might want your character's hand to go back to holding the front of a gun, or sword on their hip, or back to a steering wheel, or onto their lap etc.

Parent these *Rest* Transforms to your player or vehicle so that they move along relative to your character.

Return To Rest Move (Speed)

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a float value which determines how quickly the hand moves from any current position back to the position of the Left/Right Hand Rest Transform, when there are no objects to interact with (i.e. when no Input Handlers are enabled).

Depending on the game you are making, you may want your character's arms/hands to move swiftly, or more passively.

Return To Rest Rotate(Speed)

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a float value which determines how quickly the hand rotates from any current rotation back to the rotation of the Left/Right Hand Rest Transform, when there are no objects to interact with (i.e. when no Input Handlers are enabled).

Depending on the game you are making, you may want your character's arms/hands to rotate swiftly, or more passively.

Idle Move Speed

Necessity: *Required*

Accessibility: *Public Read Only Property*

This float value determines how fast the Left & Right Hand Helper transforms will move around when the hand is in the rest position. E.g. When the character's hand is now at rest, but on a steering wheel, we want the hands to move precisely in motion with the steering wheel, without any delays, and to do that we can set a very high value here - otherwise the hands won't be able to keep up with the steering wheel's movements, and it will no longer appear that the hand's are controlling the steering wheel.

Idle Rotate Speed

Necessity: *Required*

Accessibility: *Public Read Only Property*

This float value determines how fast the *Left & Right Hand IK Helper* transforms will rotate when the hand is in the rest position. E.g. When the character's hand is now at rest, but on a steering wheel, we want the hands to rotate precisely in motion with the steering wheel, without any delays, and to do that we can set a very high value here - otherwise the hands won't be able to keep up with the steering wheel's movements, and it will no longer appear that the hand's are controlling the steering wheel.

Hand Animator

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is the reference for the **Animator Controller** of your character model. The **Input** components (Button Input/Axis Input/ KeyCode Input etc) will use this reference to trigger any animations as needed - such as put the fingers into a palm, or stick only the thumb out, pinch etc. You don't need to have any animations at all for the fingers, but it can help make the interactions look more natural if you can see the hands up close in your game. But if it's a top down shooter or a side scrolling game, then the player likely won't be able to see the hands in much detail anyway, so the model's default animation might be perfectly fine.

Player Anims

Necessity: *Required*

Accessibility: *Public Read Only Property*

This field requires a Scriptable Object which is essentially a list of all of the animations we want to use with KINERACTIVE. This way we don't need to filter through all of the animations in the Animator Controller (See PlayerAnims SO in the manual document for more information). This field is used as reference by KINERACTIVE’s **Input** components (Axis Input component / Button Input /Keycode etc) to see which animations are available in the **Input** component’s drop down selection box.

Feet

Left Foot Helper

Necessity: *Optional*

Accessibility: *Public Read Only Property*

Essentially the same as the Left Hand Helper, except for the character’s foot. Via the IK system, the foot will follow wherever this Left Foot IK Helper Transform is located.

E.g. Place this onto a clutch pedal, foot rest plate, or rudder pedal.

This field can be left blank if not using KINERACTIVE for foot IK.

Right Foot Helper

Necessity: *Optional*

Accessibility: *Public Read Only Property*

Essentially the same as the Right Hand Helper, except for the character’s foot. Via the IK system, the foot will follow wherever this Right Foot IK Helper Transform is located.

E.g. Place this onto a gas pedal, or brake pedal, or rudder pedal.

This field can be left blank if not using KINERACTIVE for foot IK.

UI

Interaction Text

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is the reference to a Unity UI **Text (Script)** component. When we look at an interactive object, the **Input Handler** component will send a string with some usage instructions to this **Text (Script)** component. And when the player is not interacting with an object anymore, it the **Input Handler** component will disable the **Text (Script)** until another interaction takes place.

Text Background

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a reference to the background image on which the above **Text (Script)** will be placed onto - this makes the text easier to read. This field takes in a Unity UI **Image (Script)** component.

Controls Icon

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a reference to an image which will indicate a visual icon of what to press on their controller / mouse / keyboard. It is displayed at the same time as the Interaction Text to give instructions to the player on how to interact with the current object.

Crosshair

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a reference to the crosshair (or cursor) image **RawImage (Script)** component in your project. This image can be enlarged or shrunk during interactions, as well as changing the image completely. (done by **Input Handler** components). E.g. when you look at a potion, the crosshair/cursor will change to a hand icon to indicate it can be picked up.

Default Crosshair Scale

Necessity: *Required*

Accessibility: *Public Read Only Property*

Float value. The default scale of the above crosshair image, so that after an interaction has finished, the crosshair (if changed in size) can go back to the default size.

Audio

Audio Source

Necessity: *Required*

Accessibility: *Public Read Only Property*

This is a reference to the **Audio Source** component dedicated for KINERACTIVE. This **Audio Source** component will be moved around the scene, to the location of your current button/switch/dial/lever. It is moved by the currently active **Input Handler** component. So as you look from one dial to the next, you can see that this **Audio Source** component moves to the current or most recently viewed interactive object in your scene.

When a **Touchable** component plays a sound - such as the *click* of a switch, or the *thud* of a lever, it will play it from this **Audio Source** component.

Player Inputs

Player Inputs

Necessity: *Required*

Accessibility: *Public Read Only Property*

This field requires a Scriptable Object which is essentially a list of all of the inputs we want to use with KINERACTIVE. This way we don't need to filter through all of the inputs in Unity's Input Manager (See PlayerInputs SO in the manual document for more information). This field is used as reference by KINERACTIVE's **Input** components (Axis Input component / Button Input /Keycode etc) to see which buttons, keys and axis' are available in the **Input** component's drop down selection box.

Public Methods

The Kineractive Manager is created as a singleton, so running anything methods would be done from the instance property. E.g. **KineractiveManager.instance.SetCrosshairScale(7);**

SetCrosshairScale

Called by the **Input** components (Axis Input, Button Input etc) in response to controller/keyboard/mouse inputs from the player. Used to enlarge or shrink the crosshair size in order to give feedback that the player's input is being received.

SetIKTarget

Used to set which hand goes where, by moving either of the Left/Right Hand IK Helper Transforms. This is called automatically by the **Input** components (Axis Input, Button Input etc) in response to controller/keyboard/mouse inputs from the player.

Input Handlers

Input Handler

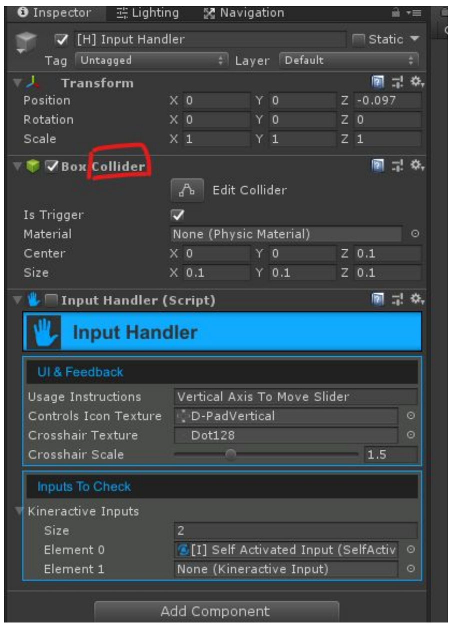
The ***Input Handler*** is the liaison between the player's input and events or actions in your game.

When it is enabled (by the **Kineractive Manager**) it checks through all player inputs that you have associated with this component. When it is disabled, no checks are done.

The **Input Handler** also passes images and text to the **UI**, which will help guide the player as to what they can do with the interactive object they are looking at.

The **Input Handler** needs to be disabled, by unchecking the checkbox in the top left corner, as we don't want to be checking for input on this object, until we are looking/aiming at it. As of v1.0 the custom editor view automatically disables all **Input Handler** components when Unity is not in "*play*" mode. And this is to prevent forgetting to disable them. However in "play" mode, they can be disabled/enabled for testing and troubleshooting.

Important: The **Input Handler** component relies on a collider or trigger - therefore it must be on the same GameObject as your collider/trigger. (Because the **Kineractive Manager** will find this **Input Handler** by shooting a raycast, and when it hits this collider, it will check to see if this collider's GameObject also contains an **Input Handler**). If you do not have a collider or trigger on the same GameObject, then the **Kineractive Manager** will not be able to find this **Input Handler**, and no interaction will be able to happen.

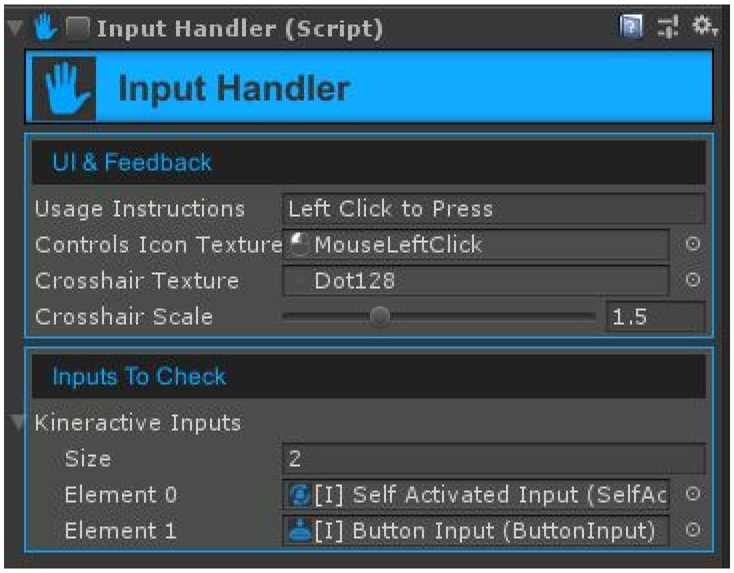


[H] Input Handler ✓
Static
Tag Untagged
Layer Default

Transform
Position X 0 Y 0 Z -0.097
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

✓ Box Collider
Edit Collider
Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 0 Z 0.1
Size X 0.1 Y 0.1 Z 0.1

Input Handler (Script)
Input Handler
UI & Feedback
Usage Instructions Vertical Axis To Move Slider
Controls Icon Texture D-PadVertical
Crosshair Texture Dot128
Crosshair Scale 1.5
Inputs To Check
Kineractive Inputs
Size 2
Element 0 [I] Self Activated Input (SelfActiv
Element 1 None (Kineractive Input)


Input Handler (Script)

Input Handler

UI & Feedback

Usage Instructions: Left Click to Press
Controls Icon Texture: MouseLeftClick
Crosshair Texture: Dot128
Crosshair Scale: 1.5

Inputs To Check

Kineractive Inputs
Size: 2
Element 0: [I] Self Activated Input (SelfAc
Element 1: [I] Button Input (ButtonInput)

Add it to the scene

To add an **Input Handler** component to your project:

- Go into the Project window and look in the Assets \ KINERACTIVE \ Scripts \ Input Handlers folder and drag and drop the **Input Handler** component into your selected GameObject (either in the Hierarchy or the Inspector).

- Or select your GameObject, and in the Inspector click the *Add Component* button, then in the popup menu select KINERACTIVE > Handler

Max Interaction Range

Usage Instructions

Necessity: *Required*

Accessibility: *Protected*

This lets you set the maximum interaction range for this specific interactive object/item. E.g. for certain switches you may want to only interact with them while the player is close to them, to ensure that the arms can reach and animate correctly. For other interactions, such as picking up an item for the floor, we can allow the interaction from a longer distance so that it's easier to pick the item up. Also make sure that your Raycast distance is set appropriately in the **Kineractive Manager** component (which should be at least as long as the object with the biggest *max interaction range*).

UI & Feedback

Usage Instructions

Necessity: *Required*

Accessibility: *Protected*

Type some brief instructions to the player - they will appear on the UI when the player looks at this object (more specifically, when this Input Handler is enabled)

Control Icon Texture

Necessity: *Required*

Accessibility: *Protected*

The *Control Icon Texture* is the image that displays when the player looks at this interactive object (more specifically, when this Input Handler is enabled). It's can be used as a non-text description of what button / joystick / key the player should press to engage this interactive object.

Crosshair Texture

Necessity: *Protected*

Accessibility: *Protected*

The crosshair on the UI will change to the texture set in this field. In most of the demo and examples in KINERACTIVE the crosshair changes to a dot, as a way to give the player feedback that they are looking at an interactive object.

Crosshair Scale

Necessity: *Protected*

Accessibility: *Public Read Only Property*

How much to enlarge or shrink the current crosshair when a connect **Input** component is pressed/clicked/axis moved.

Inputs To Check

Kineractive Inputs

Necessity: *Required*

Accessibility: *Public Read Only Property*

The **Input Handler** will execute a LateUpdate() loop where it will check each Input component listed in this array for their specified input keys/buttons/axis'. This way we can have any number and combination of **Input** component types for any interactive object. For example, if we wanted to have a button to turn a steering wheel left, and steering wheel right, we would input two

Button Input components into this list/array, and the **Input Handler** would check for them every frame.

Or if we wanted to go crazy, we could make an entire keyboard in game, and put 104 **KeyCode Input** components into the *Kineractive Inputs* array, so that it matches every key on a real keyboard.

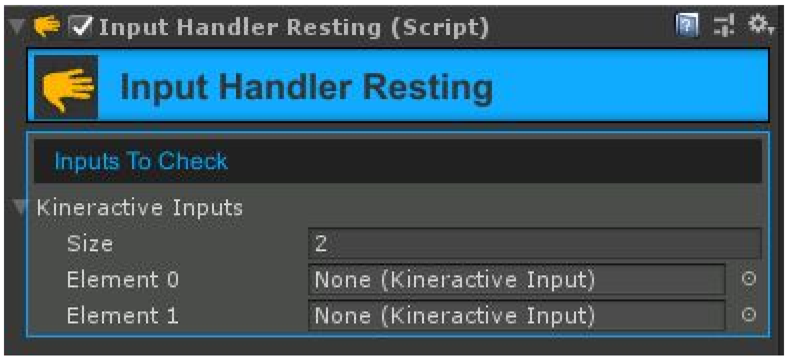
Public Methods

SetUsageInstructions

This allows the changing of usage instructions by script or events, during run-time. Useful when you want the usage *Usage Instructions* to change from: *Press To Open* to: *Press To Close* to match the open / closed state of a door or drawer.

You can use the **TextChanger** component in the Assets \ KINERACTIVE \ Scripts \ Utilities folder to help do this through events. No coding required, just create a new event, drag in the Text Changer and select the *SetUsageInstructions()* method from the event drop down box.

Input Handler Resting


Input Handler Resting (Script)

Input Handler Resting

Inputs To Check

Kineractive Inputs

Size: 2

Element 0: None (Kineractive Input)

Element 1: None (Kineractive Input)

This is an optional component, mostly for scenarios such as cockpit or car style games where you want, for example, a 3D model of a steering wheel to be moving around at all times, even when the player isn't looking at it, and you want the hands follow the movements of the steering wheel.

This allows the player's character to keep the hands on the steering wheel as the "rest position", but take them off to press buttons, and return the hands to the moving steering wheel, when they aren't being used.

See the **Steering Wheel** example scene. The **Input Hander Resting** is always active, so you can steer the steering wheel without having to look at it.

The *Kineractive Inputs* work the same as the normal **Input Handler**. The only difference here is that we don't need to set any UI options.

TIP: *Unlike the normal Input Handler, which should be disabled, this one should be enabled (for as long as you want your steering wheel to turn, which is probably always unless the car is turned off). We also do not require it to be on a collider or trigger.*

Add it to the scene

To add an **Input Handler** component to your project:

- Go into the Project window and look in the Assets \ KINERACTIVE \ Scripts \ Input Handlers folder and drag and drop the **Input Handler Resting** component into your selected GameObject (either in the Hierarchy or the Inspector).

- Or select your GameObject, and in the Inspector click the *Add Component* button, then in the popup menu select KINERACTIVE > Handler

Inputs

Introduction

The **Input** components translate real life keystrokes, button presses, and joystick movements into actions and events in the game world.

Drag and drop any combination of **Button Input / KeyCode Input / Axis Input / Self Input/ Analog Input** components into the "*Kineractive Input*s" field of the **Input Handler** on your interactive object.

Depending on how you want the player to interact with the object, you can use several **Input** components on the same **Input Handler.**

For example: To make a switch flip upwards and downwards, use individual **Input** *components* for the two actions. A **Button Input** for D-PAD up, and a **Button Input** for D-PAD down. Now the player can flick a switch up and down with the D-PAD in two directions, which is very immersive.

Or perhaps you would like the player avatar to hold an item with both hands - attach a **Self Activated Input** component for a left arm, and a second **Self Activated Input** component for the right arm, and connect them both to the *Kineractive Inputs* field on the same **Input Handler**.

All Inputs are separated into 3 sections:

1. Input Type
2. On Input Start
3. On Input End

By filling out these 3 sections we tell Unity what input we want to press for the interaction; what should happen when the input is pressed; and what should happen when the input is released.

E.g. if the player pressed the input called “Fire 1”, we want a 3D model button to be pushed in, the palm of the hand to push the 3D model button in, and when “Fire 1” is released, the hand moves away from the button, and the button pops back out again (and perhaps a door opens).

The specifics of how this interaction takes place are set up in any of the **Input** components listed in the next section, below.

Input Type is what button/key/axis the player presses on their controller/keyboard/mouse

Then we select which hand(left or right), and how fast the hand should match the target Transforms set up in the next two sections.

On Input Start, is what happens as the button/key/axis is pressed.
On Input End, is what happens when the button/key/axis is released.

In the properties of *On Input Start*, and *On Input End*, we can select the events that take place when one of these conditions is met.

- Moving and rotating the hand to a particular target transform.
- If an optional animation should play on the hand/fingers (e.g. a thumbs up)
- Unity Events that should execute

Input components can be turned on and off by using the *Bypass* checkbox on each input, or it can also be turned on/off by script or events. This is useful for when you have the same button trigger multiple things based on context, so that two animations or movements don’t happen at the same time. E.g. if you don’t want a Jet Fighter’s joystick to move if the hand is not placed on it.

Bypass is also handy for testing.

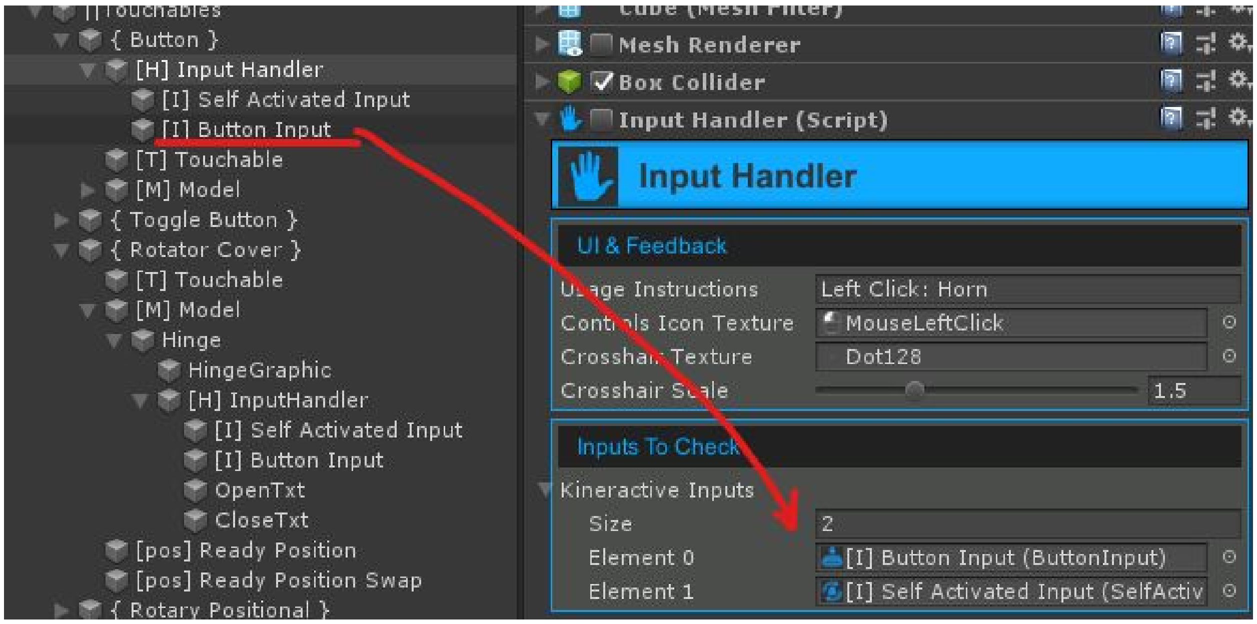
Add it to the scene

To add an **Input** component to your project:

- Go into the Project window and look in the Assets \ KINERACTIVE \ Scripts \ Inputs folder and drag and drop any of the **Input** components (such as **Axis Input**) into your selected GameObject (either in the Hierarchy or the Inspector).

- Or select your GameObject, and in the Inspector click the *Add Component* button, then in the popup menu select KINERACTIVE > Inputs

Once we are finished modifying our **Input** component, in order to actually use it, we need to place into the relevant **Input Handler** component's *Kineractive Inputs* field



```
[ ]Touchables
  { Button }
    [H] Input Handler
      [I] Self Activated Input
      [I] Button Input
    [T] Touchable
    [M] Model
  { Toggle Button }
  { Rotator Cover }
    [T] Touchable
    [M] Model
      Hinge
        HingeGraphic
        [H] InputHandler
          [I] Self Activated Input
          [I] Button Input
          OpenTxt
          CloseTxt
    [pos] Ready Position
    [pos] Ready Position Swap
  { Rotary Positional }
```

Cube (Mesh Filter)
Mesh Renderer
Box Collider
Input Handler (Script)

Input Handler

UI & Feedback

Usage Instructions	Left Click: Horn
Controls Icon Texture	MouseLeftClick
Crosshair Texture	Dot128
Crosshair Scale	1.5

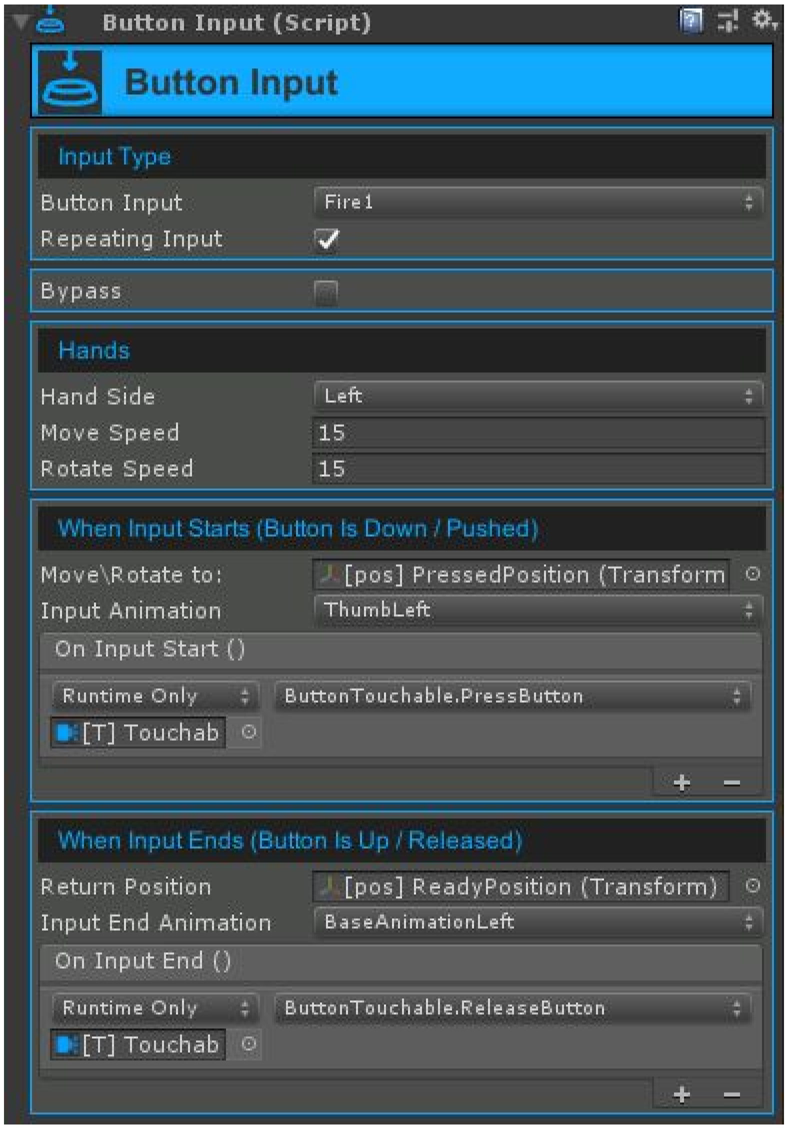
Inputs To Check

Kineractive Inputs

Size	2
Element 0	[I] Button Input (ButtonInput)
Element 1	[I] Self Activated Input (SelfActiv

This way, each interactive object has it's set of inputs and settings, each and every interactive object will have it's own unique positions for hands and different animations if we want.

Button Input


Button Input (Script)

Button Input

Input Type

Button Input: Fire1

Repeating Input: ✓

Bypass: ☐

Hands

Hand Side: Left

Move Speed: 15

Rotate Speed: 15

When Input Starts (Button Is Down / Pushed)

Move\Rotate to: [pos] PressedPosition (Transform

Input Animation: ThumbLeft

On Input Start ()

Runtime Only — ButtonTouchable.PressButton

[T] Touchab

When Input Ends (Button Is Up / Released)

Return Position: [pos] ReadyPosition (Transform)

Input End Animation: BaseAnimationLeft

On Input End ()

Runtime Only — ButtonTouchable.ReleaseButton

[T] Touchab

The **Button Input** component is used to set up interactions which you want activated with the built in **Unity Input Manager button** presses and any new ones added by you. ("Fire 1" "Fire 2" "Submit" etc)

Input Type

Button Input

Necessity: *Required*

Accessibility: *Protected*

Here we select which of **Unity's Input Manager** buttons we want to listen for. When this button is pressed, it will trigger the events in the sections below. The *Input Start* section is run when the button is pressed down. And when the button is released up, the *Input End* section is run.

Repeating Input

Necessity: *Optional*

Accessibility: *Protected*

This checkbox allows us to select whether we want the "When Input Starts" section repeated over and over, if we hold the button down. E.g. if we want to repeat a car horn sound over and over as we hold the button down. (Without this checked the car horn would only honk once).

Bypass

Necessity: *Optional*

Accessibility: *Protected (can be set via public function)*

This checkbox allows us to temporarily ignore this Input component. Useful for testing, or in certain cases in your game, you may not want the player to be able to interact with certain objects or during cutscenes, or perhaps you want to "lock" a switch until a cover on it is opened.

Hands

Hand Side

Necessity: *Required*

Accessibility: *Protected*

Choose which hand/arm we are going to move - either the left or right. This field determines whether the **Kineractive Manager** component moves the **Left Hand IK Helper** Transform or the **Right Hand IK Helper** Transform

---

Move Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands /arms will move to their new position (as set in the *Move\Rotate* to sections below)

Rotate Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands / arms will rotate to their new rotation (as set in the *Move\Rotate* to sections below)

When Input Starts

When the button (as selected in the *Button Input* drop down list) is pressed down, the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input Start()* list will be run.

For example, if we want to use the character's thumb to press a switch, we would set the *Move\Rotate* to an empty Transform near the in-game switch so that the hand can be moved to that area. We would select the "Thumb" animation, and add an event into the *On Input Start()* area which runs the "PressSwitch" method on the switch's GameObject.

Move\Rotate to

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will move to - whatever position and rotation the Transform in this field is set to, our hand will copy (via the Kineractive Manager component). So place this empty Transform in a spot where suitable to give your hand animation a real and natural motion and look.

Input Animation

Necessity: *Optional*

Accessibility: *Protected*

Here we can optional animation to play, other than the one that is currently playing on the base model. For example, we may want our thumb or index finger to stick out to press a switch, or perhaps we want a flat palm to have our biosigns read by a hand print reader etc.

On Input Start

Necessity: *Optional*

Accessibility: *Protected*

This is where any events happen as the button is pressed. E.g. if we want a switch to be flicked up or down, then we would add a new event here, and drag in the switch's script, and then run the switch's FlickUp() or FlickDown() method.

To begin with, you most likely will want to use any of the **Touchable** components here, as they have already been setup with these events in mind. For example, to have a switch flick up and down, you could use the **Rotator** Touchable component to perform this switch flick animation.

When Input Ends

When the button (as selected in the *Button Input* drop down list) is released(up), the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input End()* list will be run.

For example, the character's thumb may have been sticking out to press a switch. So here we would set the animation to return to the default / base animation of the character, then we also want the switch to return back to the neutral position, so we would add that an event in the *On Input End()* section. And lastly, we would set where we want the hand to return to when we let go of our button - either back to the rest position, or perhaps the switch's ready position - a Transform located near the switch, but not actually pressing it.

Return Position

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will return to when the button is released. Ideally make this the ready position (same Transform as you might be using the Self Activated Input component) or make this the Rest Position. But it can be anything as you require - wherever you want the hand to end up, after the player releases this input button.

Input End Animation

Necessity: *Required*

Accessibility: *Protected*

Please select the default / base animation of your character (or to an empty Animation node on the Animator Controller). We want to use this field to return the left or right hand back to a neutral animation in most case - perhaps in some cases you could have it so that the character shake's their hand, if they touched something hot etc

---

On Input End

Necessity: *Optional*

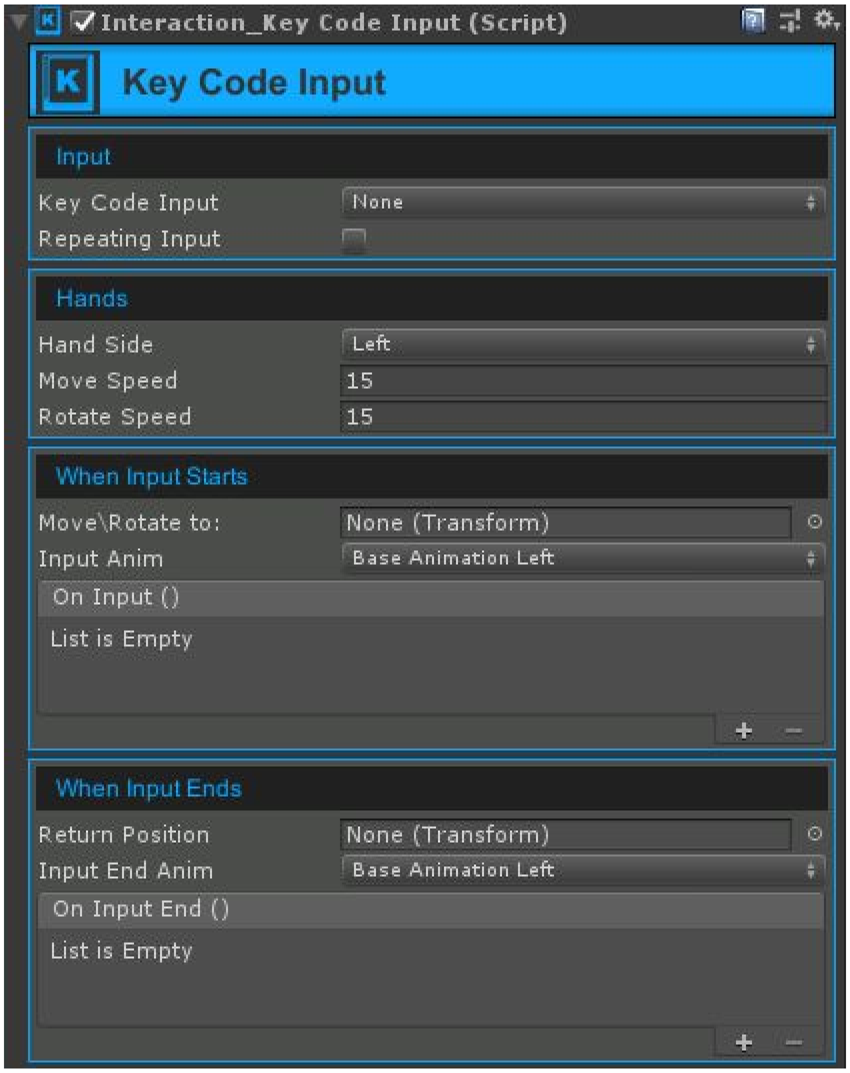
Accessibility: *Protected*

We can use this event section to reset any buttons or switches if required. For example, if we press a switch, and we want the switch to return back to its initial position (as if it has a spring on it), then we can place that event here.

Or use this section to run a script that checks whether the player has finished interacting with this button - e.g. for a tutorial or monster spawn etc

---

KeyCode Input


Interaction_Key Code Input (Script)

Key Code Input

Input

Key Code Input: None

Repeating Input: ☐

Hands

Hand Side: Left

Move Speed: 15

Rotate Speed: 15

When Input Starts

Move\Rotate to: None (Transform)

Input Anim: Base Animation Left

On Input ()

List is Empty

When Input Ends

Return Position: None (Transform)

Input End Anim: Base Animation Left

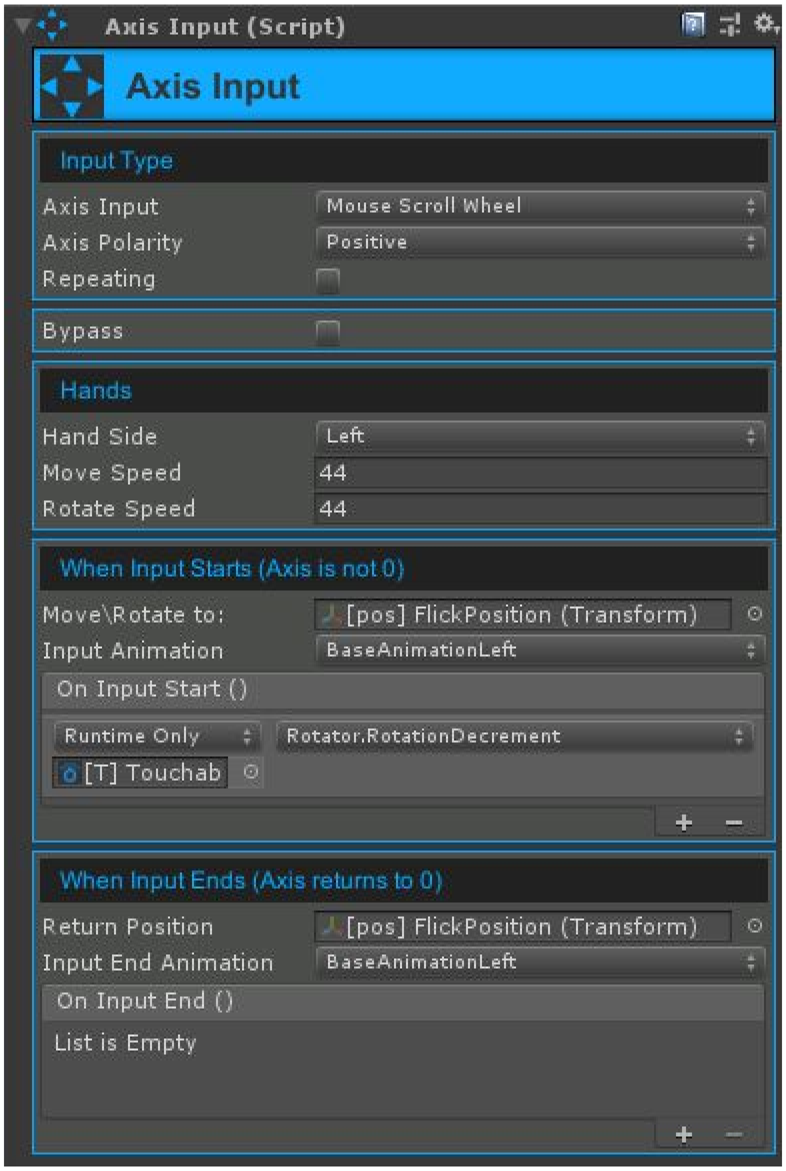
On Input End ()

List is Empty

Essentially the same as **Button Input** however allowing for direct mapping to the keyboard. I'd recommend avoiding using this over the **Button Input** component, because it bypasses the **Unity Input Manager,** as well as the PlayerInputs Scriptable Objects, so this might make it more difficult to make easy changes later on, as you won't have a centralised area to make those changes and will need to make the change on every single **KeyCode Input** component in your entire project. But it is here if you need to use direct keyboard mapping.

All of the fields are identical to those in Button Input (above). Please check above for information regarding each field and drop down box.

Axis Input


Axis Input (Script)

Axis Input

Input Type

Axis Input: Mouse Scroll Wheel
Axis Polarity: Positive
Repeating: ☐

Bypass: ☐

Hands

Hand Side: Left
Move Speed: 44
Rotate Speed: 44

When Input Starts (Axis is not 0)

Move\Rotate to: [pos] FlickPosition (Transform)
Input Animation: BaseAnimationLeft

On Input Start ()
Runtime Only | Rotator.RotationDecrement
[T] Touchab

When Input Ends (Axis returns to 0)

Return Position: [pos] FlickPosition (Transform)
Input End Animation: BaseAnimationLeft

On Input End ()
List is Empty

Essentially the same as *Button Input* but allows the game designer to select from the **Unity Input Manager axes** instead. *E.g.* by default we can choose either of Unity's *Horizontal* or *Vertical* axes as the input. Further, we must choose a *positive* or *negative* polarity to select whether we want to listen for an Up or Down input, or Left or Right input.

E.g. choosing *Vertical* axis, and *Positive polarity* - this will listen for a **W key, Up Arrow** press or **Analog Stick Up** movement.

If we were to use the *Negative* polarity instead, it would listen for a **Downward Analog Stick** movement, **S key** or **Down Arrow** press.

If you need analog based input, then please use the **Analog Input** component instead.

Axis Input works as a two state button (pressed and released). We will enable the "*pressed*" or "*When Input Starts*" state if we move the axis to any value greater than 0. If the value goes back to 0, after being greater than 0, then this counts and "*released*" state or "*When Input Ends*" state.

We are making the axis mimic button presses, and you can think of **Axis Input** as using the *D-PAD* or *Arrow Keys or Scroll Wheel*, rather than an *Analog Stick* (those can be used too, however will only return pressed and released states, and not the analog value if we are using the **Axis Input** component - for true analog input values, please use the **Analog Input** component).

Because there is a Positive and Negative range/polarity for each selected axis, if we want to move a lever, slider or switch in both directions, then we can use two **Axis Input** components - one for each direction - e.g. using the Mouse Scroll Wheel axis, we would use two components to read the down and up direction separately.

(in fact we can even use 4 **Axis Inputs** if we want a 4 way direction switch in game - such as an ingame representation of a D-PAD style control or a HAT switch as found on many flight sticks)

Input Type

Axis Input

Necessity: *Required*

Accessibility: *Protected*

Here we select which of **Unity's Input Manager** axes we want to listen for. When this axis is pressed engage (by moving it beyond zero), it will trigger the events in the sections below. The *Input Start* section is run when the axis is no longer zero. And when the axis is returned to zero after not being zero in the last frame, the *Input End* section is run.

Axis Polarity

Necessity: *Required*

Accessibility: *Protected*

We've selected which Axis we want to use in the section above, now we must choose whether we want to listen for this axis' positive or negative range.

In Unity, all axes range from -1.0 to 1.0, and here we can choose if we want to listen for any axis inputs above zero, or below zero. (e.g. for the vertical axis, any down stick movements, or S key presses would be below zero, and up movements mean values above zero). In a way we are choosing which direction to listen for - whether it is in the up direction, or is it the down direction.

(Or if using the horizontal axis, it would be left (between -1.0 & 0) or right (between 0 & 1.0).

Any value other than zero in the axis counts as a "press"

Repeating Input

Necessity: *Optional*

Accessibility: *Protected*

This checkbox allows us to select whether we want the "When Input Starts" section repeated over and over, if we hold the axis direction. E.g. if we want to repeat a car horn sound over and over as we hold the axis in a direction. (Without this checked the car horn would only honk once).

Bypass

Necessity: *Optional*

Accessibility: *Protected (can be set via public function)*

This checkbox allows us to temporarily ignore this Input component. Useful for testing, or in certain cases in your game, you may not want the player to be able to interact with certain objects or during cutscenes, or perhaps you want to "lock" a switch until a cover on it is opened.

Hands

Hand Side

Necessity: *Required*

Accessibility: *Protected*

Choose which hand/arm we are going to move - either the left or right. This field determines whether the **Kineractive Manager** component moves the **Left Hand IK Helper** Transform or the **Right Hand IK Helper** Transform

---

Move Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands /arms will move to their new position (as set in the *Move\Rotate* to sections below)

Rotate Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands / arms will rotate to their new rotation (as set in the *Move\Rotate* to sections below)

When Input Starts

When the axis is no longer zero it is considered as “pressed down”, and the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field’s Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input Start()* list will be run.

For example, if we want to use the character’s thumb to press a switch, we would set the *Move\Rotate* to an empty Transform near the in-game switch so that the hand can be moved to that area. We would select the “Thumb” animation, and add an event into the *On Input Start()* area which runs the “PressSwitch” method on the switch’s GameObject.

Move\Rotate to

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will move to - whatever position and rotation the Transform in this field is set to, our hand will copy (via the Kineractive Manager component). So place this empty Transform in a spot where suitable to give your hand animation a real and natural motion and look.

Input Animation

Necessity: *Optional*

Accessibility: *Protected*

Here we can optional animation to play, other than the one that is currently playing on the base model. For example, we may want our thumb or index finger to stick out to press a switch, or perhaps we want a flat palm to have our biosigns read by a hand print reader etc.

On Input Start

Necessity: *Optional*

Accessibility: *Protected*

This is where any events happen as the axis is to any value other than zero.

E.g. if we want a switch to be flicked up or down, then we would add a new event here, and drag in the switch's script, and then run the switch's FlickUp() or FlickDown() method.

To begin with, you most likely will want to use any of the **Touchable** components here, as they have already been setup with these events in mind. For example, to have a switch flick up and down, you could use the **Rotator** Touchable component to perform this switch flick animation.

When Input Ends

When the axis (as selected in the *Axis Input* drop down list) is released (when the axis value is equal to 0 after not being 0 in the previous frame), the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input End()* list will be run.

For example, the character's thumb may have been sticking out to press a switch. So here we would set the animation to return to the default / base animation of the character, then we also want the switch to return back to the neutral position, so we would add that an event in the *On Input End()* section. And lastly, we would set where we want the hand to return to when we let go of our button - either back to the rest position, or perhaps the switch's ready position - a Transform located near the switch, but not actually pressing it.

Return Position

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will return to when the axis is released. Ideally make this the ready position (same Transform as you might be using the Self Activated Input component) or make this the Rest Position. But it can be anything as you require - wherever you want the hand to end up, after the player releases this axis - i.e when the axis is back to 0.

Input End Animation

Necessity: *Required*

Accessibility: *Protected*

Please select the default / base animation of your character (or to an empty Animation node on the Animator Controller). We want to use this field to return the left or right hand back to a neutral animation in most case - perhaps in some cases you could have it so that the character shake's their hand, if they touched something hot etc

---

On Input End

Necessity: *Optional*

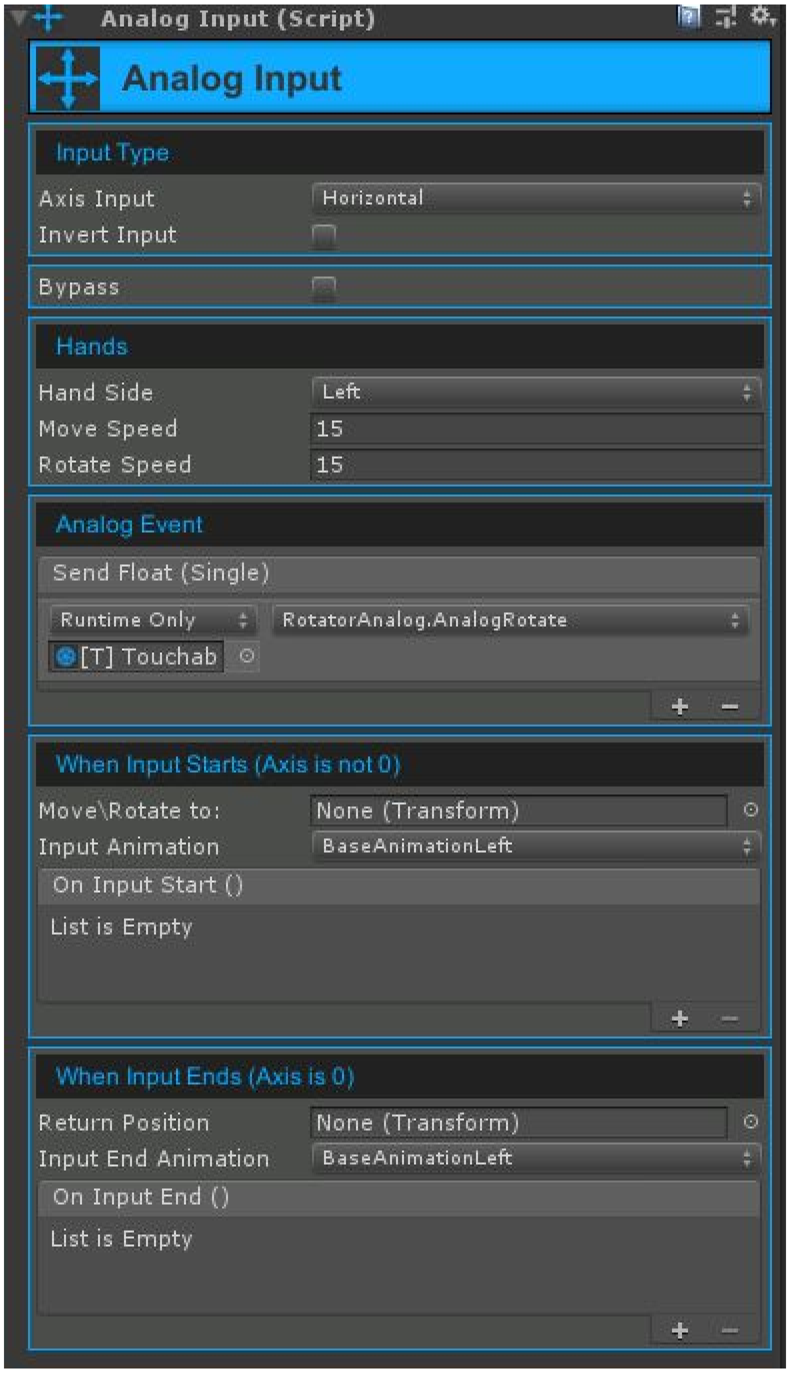
Accessibility: *Protected*

We can use this event section to reset any buttons or switches if required. For example, if we press a switch, and we want the switch to return back to its initial position (as if it has a spring on it), then we can place that event here.

Or use this section to run a script that checks whether the player has finished interacting with this button - e.g. for a tutorial or monster spawn etc

---

Analog Input


Analog Input (Script)

Analog Input

Input Type

Axis Input: Horizontal

Invert Input: ☐

Bypass: ☐

Hands

Hand Side: Left

Move Speed: 15

Rotate Speed: 15

Analog Event

Send Float (Single)

Runtime Only | RotatorAnalog.AnalogRotate

[T] Touchab

When Input Starts (Axis is not 0)

Move\Rotate to: None (Transform)

Input Animation: BaseAnimationLeft

On Input Start ()

List is Empty

When Input Ends (Axis is 0)

Return Position: None (Transform)

Input End Animation: BaseAnimationLeft

On Input End ()

List is Empty

The **Analog Input** component returns the value in the full range of a select axis. If we select the Horizontal axis, and we were to use an analog stick like that of an XBOne / PS4 / N-switch, we can receive the full range between -1.0 and 1.0

By using the Analog Input component we can set how fast dials or levers turn, based on how far we push the analog stick on our real life control pad. Or we can make an in-game joystick go to the exact position as our analog stick - giving us 1:1 representation of our analog control pad in game.

To make the best use of this component, we should pair it with a **Touchable** component that is also analog based - such as the **MoverAnalog** or **RotatorAnalog** components as they contain methods called *SetAbsolutePosition()* and *AnalogMove()* or *SetAbsoluteRotation()* and *AnalogRotate()*, which know how to handle the values coming from SendFloat event.

Input Type

Axis Input

Necessity: *Required*

Accessibility: *Protected*

Here we select which of **Unity's Input Manager** axes we want to listen to.

Bypass

Necessity: *Optional*

Accessibility: *Protected (can be set via public function)*

This checkbox allows us to temporarily ignore this Input component. Useful for testing, or in certain cases in your game, you may not want the player to be able to interact with certain objects or during cutscenes, or perhaps you want to "lock" a switch until a cover on it is opened.

Hands

Hand Side

Necessity: *Required*

Accessibility: *Protected*

Choose which hand/arm we are going to move - either the left or right. This field determines whether the **Kineractive Manager** component moves the **Left Hand IK Helper** Transform or the **Right Hand IK Helper** Transform

Move Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands /arms will move to their new position (as set in the *Move\Rotate* to sections below)

Rotate Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands / arms will rotate to their new rotation (as set in the *Move\Rotate* to sections below)

Analog Event

Send Float (Single)

Necessity: *Required*

Accessibility: *Protected*

This is the main feature of this component.

While the **Input Handler** linked to this **Analog Input** component is active, this **Analog Input** component will continually send the float value of this axis to the methods added in the SendFloat(Single) event.

The float value sent is between -1.0 and 1.0 so any methods receiving this float value will need to know how to use or scale this value range. (The Analog Mover and Analog Rotater components have methods to handle this value range correctly).

When Input Starts

When the axis not zero the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input Start()* list will be run.

For example, if we want to use the character's thumb to press a switch, we would set the *Move\Rotate* to an empty Transform near the in-game switch so that the hand can be moved to that area. We would select the "Thumb" animation, and add an event into the *On Input Start()* area which runs the "PressSwitch" method on the switch's GameObject.

Move\Rotate to

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will move to - whatever position and rotation the Transform in this field is set to, our hand will copy (via the Kineractive Manager component). So place this empty Transform in a spot where suitable to give your hand animation a real and natural motion and look.

Input Animation

Necessity: *Optional*

Accessibility: *Protected*

Here we can optional animation to play, other than the one that is currently playing on the base model. For example, we may want our thumb or index finger to stick out to press a switch, or perhaps we want a flat palm to have our biosigns read by a hand print reader etc.

On Input Start

Necessity: *Optional*

Accessibility: *Protected*

This is where any events happen as the axis is to any value other than zero.

E.g. if we want a switch to be flicked up or down, then we would add a new event here, and drag in the switch's script, and then run the switch's FlickUp() or FlickDown() method.

To begin with, you most likely will want to use any of the **Touchable** components here, as they have already been setup with these events in mind. For example, to have a switch flick up and down, you could use the **Rotator** Touchable component to perform this switch flick animation.

When Input Ends

When the axis is zero the following actions will happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input End()* list will be run.

For example, the character's thumb may have been sticking out to press a switch. So here we would set the animation to return to the default / base animation of the character, then we also want the switch to return back to the neutral position, so we would add that an event in the *On Input End()* section. And lastly, we would set where we want the hand to return to when we let go of our button - either back to the rest position, or perhaps the switch's ready position - a Transform located near the switch, but not actually pressing it.

Return Position

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will return to when the axis is 0. Ideally make this the ready position (same Transform as you might be using the Self Activated Input component) or make this the Rest Position. But it can be anything as you require - wherever you want the hand to end up, after the player releases this axis - i.e when the axis is back to 0.

Input End Animation

Necessity: *Required*

Accessibility: *Protected*

Please select the default / base animation of your character (or to an empty Animation node on the Animator Controller). We want to use this field to return the left or right hand back to a neutral animation in most case - perhaps in some cases you could have it so that the character shake's their hand, if they touched something hot etc

---

On Input End

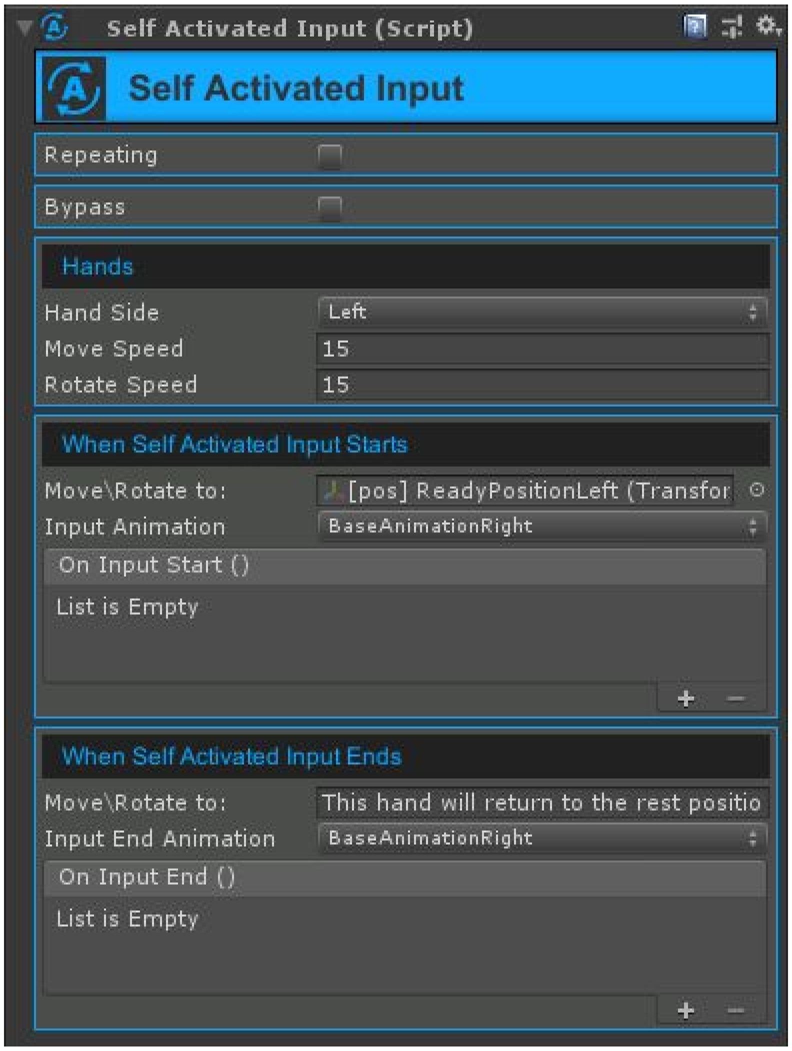
Necessity: *Optional*

Accessibility: *Protected*

We can use this event section to reset any buttons or switches if required. For example, if we press a switch, and we want the switch to return back to its initial position (as if it has a spring on it), then we can place that event here.

Or use this section to run a script that checks whether the player has finished interacting with this button - e.g. for a tutorial or monster spawn etc

Self Activated Input


Self Activated Input (Script)

Self Activated Input

Repeating	☐
Bypass	☐

Hands

Hand Side	Left
Move Speed	15
Rotate Speed	15

When Self Activated Input Starts

Move\Rotate to:	[pos] ReadyPositionLeft (Transfor
Input Animation	BaseAnimationRight

On Input Start ()

List is Empty

When Self Activated Input Ends

Move\Rotate to:	This hand will return to the rest positio
Input End Animation	BaseAnimationRight

On Input End ()

List is Empty

Self Activated Input can be thought of as automatic input. As soon as the *Interactive Trigger* is enabled, then *Self Input* executes the movement, rotation, animations and events set in the middle section “When Trigger Enables”

And when the *Interactive Trigger* disables again, the “When Trigger Disables” events run.

This is useful for when you want the player avatar’s hands to get into a “ready” position before pressing a button or switch, so that it doesn’t look like the avatar is slapping switches. Instead it gives the impression that the player avatar reaches out more gracefully before flicking a switch.

Repeating Input

Necessity: *Optional*

Accessibility: *Protected*

This checkbox allows us to select whether we want the “When Input Starts” section repeated over and over, while this component’s Input Handler is active.

Bypass

Necessity: *Optional*

Accessibility: *Protected (can be set via public function)*

This checkbox allows us to temporarily ignore this Input component. Useful for testing, or in certain cases in your game, you may not want the player to be able to interact with certain objects or during cutscenes, or perhaps you want to “lock” a switch until a cover on it is opened.

Hands

Hand Side

Necessity: *Required*

Accessibility: *Protected*

Choose which hand/arm we are going to move - either the left or right. This field determines whether the **Kineractive Manager** component moves the **Left Hand IK Helper** Transform or the **Right Hand IK Helper** Transform

Move Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands /arms will move to their new position (as set in the *Move\Rotate* to sections below)

Rotate Speed

Necessity: *Required*

Accessibility: *Protected*

How fast the hands / arms will rotate to their new rotation (as set in the *Move\Rotate* to sections below)

When Input Starts

When the **Input Handler** this **Self Activated Input** component is linked to, and the following actions will automatically happen: The **Kineractive Manager** will be told to move the **IK Helper** transform for the corresponding arm to the *Move\Rotate* field's Transform position and rotation. The select animation will play on the hand, and all of the events added to the *On Input Start()* list will be run.

For example, if we want the character's hand to move to a "ready position" to get ready to press a switch, we would set the *Move\Rotate* to an empty Transform near the in-game switch so that while the game wait's for the player's input, the hand is waiting in a natural spot, ready to press. This really helps sell the hand movements overall, because if we were to move the hand from the rest position (e.g. from the player's lap or gun), it can look like that the hands are slapping the switches instead of move toward them normally. Typically slapping is not what we want so this is one way to make the interactions look more natural.

Move\Rotate to

Necessity: *Required*

Accessibility: *Protected*

This is where our hand will move to - whatever position and rotation the Transform in this field is set to, our hand will copy (via the Kineractive Manager component). So place this empty Transform in a spot where suitable to give your hand animation a real and natural motion and look.

Input Animation

Necessity: *Optional*

Accessibility: *Protected*

Here we have an optional animation to play, other than the one that is currently playing on the base model. For example, we may want our thumb or index finger to stick out in anticipation of flicking a switch.

On Input Start

Necessity: *Optional*

Accessibility: *Protected*

This is where any events happen as soon as the **Input Handler** linked to this component is activated.

E.g. we might want to set an event to a tutorial script to tell the tutorial to perform the next step, or maybe we want the character's hand push open a door as soon as they look at it.

When Input Ends

When the Input Handler linked to this input is disabled, it runs the following actions: The **Kineractive Manager** will be told to move the **IK Helper** transform will return to the rest position (set in the Kineractive Manager). The selected animation will play on the hand, and all of the events added to the *On Input End()* list will be run.

For example, the character's hand may have been near a switch in a "ready position" waiting for player input. So now if the player looks away from the switch, the hand will move from the *ready position* back to the *rest position*. This is also a good section to make sure any animations on the hand are set back to the base animation of our character.

Return Position

Necessity: *Automatic*

Accessibility: *Protected*

The return position for the **Self Activated Input** is always the Rest Position - because when the interaction ends (i.e. we look away from the a switch) then we want the hand to return to the rest position - whether it's back to holding a gun, or a sword, or steering wheel. It is hard coded for convenience. (of course you can always extend the **Self Activated Input** and change this if needed, and please let me know if you do, I'd like to hear about every use case).

Input End Animation

Necessity: *Required*

Accessibility: *Protected*

Please select the default / base animation of your character (or to an empty Animation node on the Animator Controller). We want to use this field to return the left or right hand back to a neutral animation in most case - perhaps in some cases you could have it so that the character shake's their hand, if they touched something hot etc

---

On Input End

Necessity: *Optional*

Accessibility: *Protected*

For the Self Activated Input component, this is a good place to put in button reset events - for example if we are holding down a key for pressing a car horn button, and the player then looks away from the car horn button, we can run a "unclick()" method on the button, so that the car horn sound stops playing, and the button pop's back out to its default position.

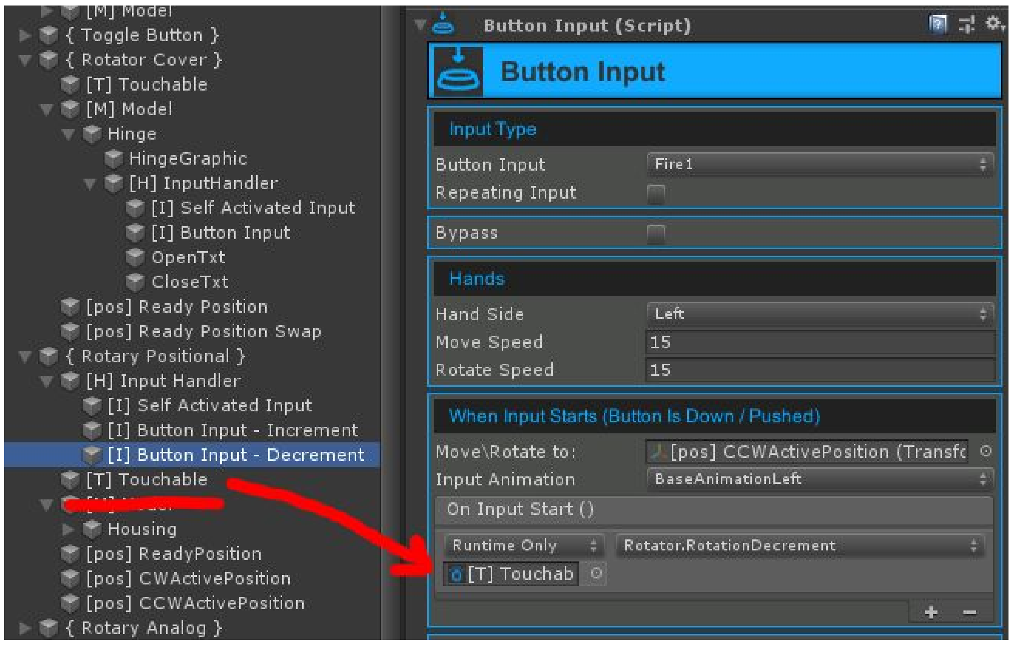
Touchables

Touchables are objects that the player can interact with, and the player's character will reach out to touch and manipulate. The in-game-world controls, buttons, keypads, switches, throttles and sticks, levers, etc. All of these examples can be made by just a handful of touchables configured in the right way. You can also create your own, by extending the *Touchable* base class.

You can also do more than just create controls with Touchables - because they move and rotate, you can repurpose them for (or extend them as) moving platforms or doors in your level.

Add it to the scene

You can place **Touchable** components just about anywhere, but it makes sense to place them in a folder GameObject parent, to it grouped with your Inputs, Input Handler and 3D model as well. Place it on any gameObject in your Hierarchy structure so that it makes sense to you. Then just use the events UI system in the **Inspector** from the relevant **Input** components to invoke the methods from these **Touchable** components.



```
    [M] Model
{ Toggle Button }
{ Rotator Cover }
  [T] Touchable
  [M] Model
    Hinge
      HingeGraphic
      [H] InputHandler
        [I] Self Activated Input
        [I] Button Input
        OpenTxt
        CloseTxt
  [pos] Ready Position
  [pos] Ready Position Swap
{ Rotary Positional }
  [H] Input Handler
    [I] Self Activated Input
    [I] Button Input - Increment
    [I] Button Input - Decrement
  [T] Touchable
  [M] Model
    Housing
  [pos] ReadyPosition
  [pos] CWActivePosition
  [pos] CCWActivePosition
{ Rotary Analog }
```

Button Input (Script)

Button Input

Input Type

Button Input	Fire1
Repeating Input	☐
Bypass	☐

Hands

Hand Side	Left
Move Speed	15
Rotate Speed	15

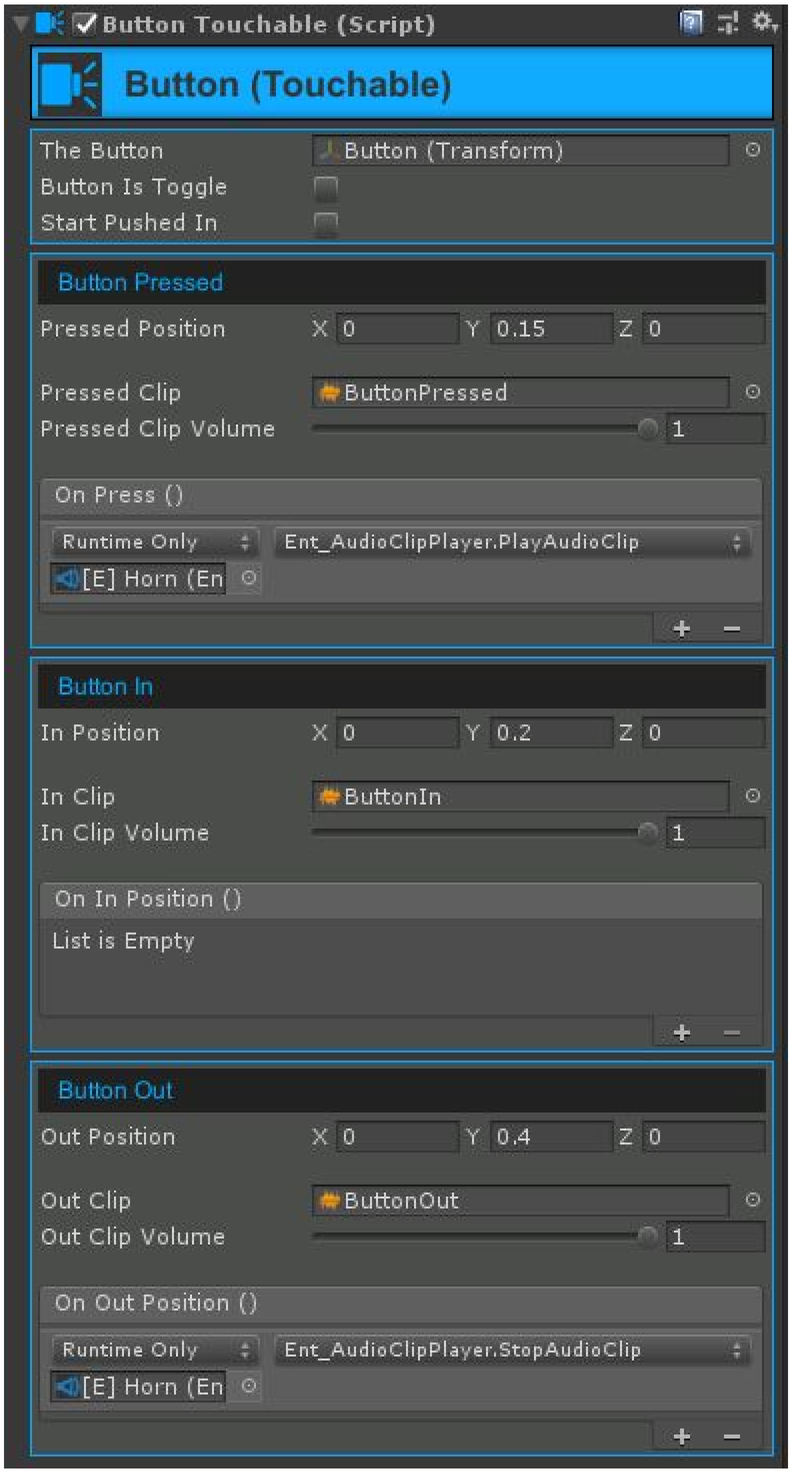
When Input Starts (Button Is Down / Pushed)

Move\Rotate to:	[pos] CCWActivePosition (Transfc
Input Animation	BaseAnimationLeft

On Input Start ()

Runtime Only	Rotator.RotationDecrement
[T] Touchab	

Button (Touchable)



Button Touchable (Script)

Button (Touchable)

The Button: Button (Transform)
Button Is Toggle: ☐
Start Pushed In: ☐

Button Pressed

Pressed Position: X 0 Y 0.15 Z 0
Pressed Clip: ButtonPressed
Pressed Clip Volume: 1

On Press ()
Runtime Only | Ent_AudioClipPlayer.PlayAudioClip
[E] Horn (En

Button In

In Position: X 0 Y 0.2 Z 0
In Clip: ButtonIn
In Clip Volume: 1

On In Position ()
List is Empty

Button Out

Out Position: X 0 Y 0.4 Z 0
Out Clip: ButtonOut
Out Clip Volume: 1

On Out Position ()
Runtime Only | Ent_AudioClipPlayer.StopAudioClip
[E] Horn (En

The **Button Touchable** is the representation of a button object in your 3D world. It is the interactive object your character will touch with their hand (not to be confused with the **Button Input** component, or a **Unity** input button)

The **Button Touchable** component has **3** states, *Pressed*, *In*, and *Out*. This allows us to configure it as a keyboard style key (with 2 states *Pressed* & *Out*) or like a toggle button, such as the PAUSE button on a radio tape deck which has all 3 states:

- *Pressed* position is the temporary state while our finger is on the pause button, pushing it down.
- When we release our finger, the PAUSE button goes into the *In* position, pausing the tape.

- If we press the PAUSE button again, it will go into the *Pressed* state once more, but on release it will now toggle to the *Out* state, unpausing the tape.

The Button

Necessity: *Required*

Accessibility: *Protected*

This field should be set to Transform containing the 3D model of the actual button - the part that we want to move in and out, as it is pushed by the in game character's fingers/hands.

Button Is Toggle

Necessity: *Optional*

Accessibility: *Protected*

Changes the button between type buttons styles:

- **Regular style:** We push the button, and it pops out as soon as we let go. Switching only between the *Pressed* and *Out* states.

- **Toggle Style:** We push the button, and when we let go, it is set to the *In* position, if we push the button again, it is then set to the *Out* position. So it changes between *Pressed > In > Pressed > Out* over and over. It is a 3 state button, with the *Pressed* state being only a temporary state for a nice effect - the *In*, and *Out* states in this style, are where you would put your events.

Start Pushed In

Necessity: *Optional*

Accessibility: *Protected*

Tick this if we want the button to start in the *In* position. (note: it will not run the events or play the audio clips from that section when the scene is played - only the Transform's position is changed)

Button Pressed

Pressed Position

Necessity: *Required*

Accessibility: *Protected*

This should be set to the furthest inward position we want the button to reach. This will illustrate to the player that we are pushing the button. And when it is released, it will outwards into the *In* or *Out* states.

Note that we can move the button Transform in any direction by using the XYZ coordinates. The movement is done in local space (relative to the rest of the button model). So the button doesn't strictly have to be a typical In-Out button. But it can be changed to slide up or down, or left or right etc, giving us instantly a lot of different switch and button styles.

Pressed Clip

Necessity: *Optional*

Accessibility: *Protected*

This audio clip will play when the button reaches the *Pressed* state. The audio clip is played from the Kineractive **Audio Source** component as is assigned in the **Kineractive Manager** component.

Pressed Clip Volume

Necessity: *Optional*

Accessibility: *Protected*

Set the volume of the above audio clip here.

On Press()

Necessity: *Optional*

Accessibility: *Protected*

When the *Pressed* state is reached, any methods placed into this event will be executed. (e.g. "play car horn");

Button In

The *In* Position is not used in the non-toggle mode. This section is only used if the *Button Is Toggle* checkbox is ticked.

In Position

Necessity: *Required*

Accessibility: *Protected*

This where the button position will be set to, after the *Pressed* position ends (in toggle mode).

This should be set to the second most inward position of the button (if using a standard in-out button style).

In Clip

Necessity: *Optional*

Accessibility: *Protected*

The audioclip which is played by the Kineractive Manager's **Audio Source** component. It should be a nice clicking sound.

In Clip Volume

Necessity: *Required*

Accessibility: *Protected*

The volume of the clip set above.

On In Position

Necessity: *Optional*

Accessibility: *Protected*

When the *In* state is reached, any methods placed into this event will be executed. (e.g. “turn on headlights”);

(It is optional, but not much will happen if you don’t assign anything here!)

Button Out

Out Position

Necessity: *Required*

Accessibility: *Protected*

The outermost position we want the button to be out - this is the position the button will be in when it is not being pressed.

Out Clip

Necessity: *Required*

Accessibility: *Protected*

The click sound of your button when it moves to the *Out* position

---

Out Clip Volume

Necessity: *Required*

Accessibility: *Protected*

The volume of the sound clip set above.

On Out Position

Necessity: *Required*

Accessibility: *Protected*

When the *Out* state is reached, any methods placed into this event will be executed. (e.g. “stop the car horn sound from playing immediately”);

Public Methods

These methods should be used as part of invents in your **Input** components. E.g. Your **KeyCode Input** component can add the *PressButton()* method in it’s events, so that when you press a key on your keyboard, the *PressButton()* method is invoked.

PressButton()

This will initiate the button press. You would put this in the *When Input Starts* event in one of your **Input** components.

ReleaseButton()

This will initiate the button release. You would put this in the *When Input Ends* event in one of your **Input** components.

UnClickButton()

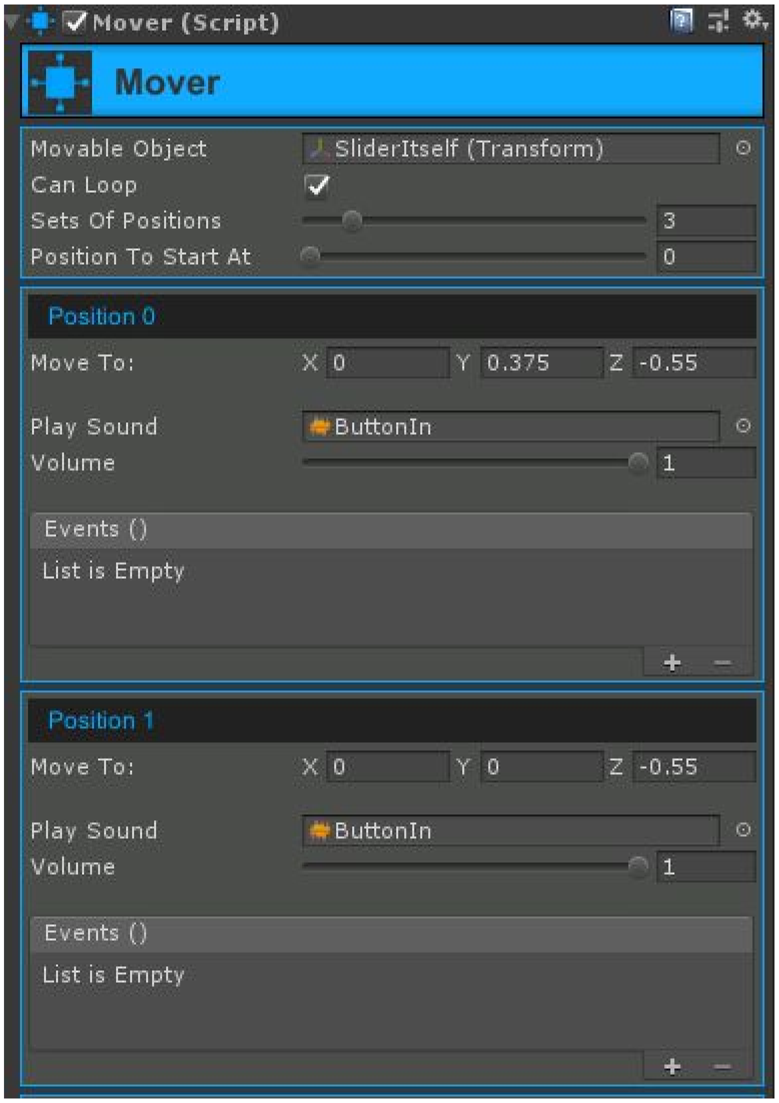
This will return the button to it's Out position, but only if the button is currently being pressed. This is useful to put in the **Self Activated Input** components *When Input Ends* event - because if you are holding down a horn button, and you look away from the horn button, we want that horn button to pop back out and stop playing. This would only be used if you are not creating a toggle button.

Public Properties

Button_State

You can use this to check what state the button is in.

Mover


Mover (Script)

Mover

Movable Object: SliderItself (Transform)
Can Loop: ✓
Sets Of Positions: 3
Position To Start At: 0

Position 0
Move To: X 0 Y 0.375 Z -0.55
Play Sound: ButtonIn
Volume: 1
Events ()
List is Empty

Position 1
Move To: X 0 Y 0 Z -0.55
Play Sound: ButtonIn
Volume: 1
Events ()
List is Empty

The **Mover** Touchable component is similar to a **Button** Touchable, in that it moves between various positions and when each state or position is reached, it will play a sound, and run the events associated with that position or state.

However the difference is that the **Mover** component has up to 10 states/positions it can move between (technically unlimited amount of states if extended to do so). So this allows us to do things that the button cannot, such as:

Creating a slider with 10 different positions. Or a gear lever in a Japanese make car where the gear selector is able to travel forward/backwards, but also sideways movements to go around notches when changing from Park to Drive positions.

We can also choose to move between the positions sequentially up or down, or go directly to a set position, by using the various methods outlined below.

Moveable Object

Necessity: *Required*

Accessibility: *Protected*

The Transform which want to move around - relative to the position of it's parent object.

Can Loop

Necessity: *Optional*

Accessibility: *Protected*

When incrementing or decrementing positions - are we allowed to go back to around to the first position if the last position has been reached (and vice versa).

Sets of Positions

Necessity: *Required*

Accessibility: *Protected*

How many positions we want this Touchable to have. Select between 2 and 10. Two positions if you want to make a simple button, or up to 10 positions if you want to make a more complicated slider style Touchable.

Position To Start At

Necessity: *Optional*

Accessibility: *Protected*

Which of our positions should the *Movable Object* field's Transform start in (note it will not run the events or play the sounds, only the Transform's position is changed)

Position

As we add positions, each new position has its own set of sounds, positions and events to invoke as each position is reached.

Move To:

Necessity: *Required*

Accessibility: *Protected*

The location the *Movable Object* will move to when we are in this position #

Play Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound played when this position is reached (played by the Kinerative Manager's **Audio Source** component)

Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the above position.

Events()

Necessity: *Optional*

Accessibility: *Protected*

When this position i reached it will invoke all of the methods in this event.

Public Methods

These methods should be used as part of invents in your **Input** components. E.g. Your **KeyCode Input** component can add the *PositionIncrement()* method in it's events, so that when you press a key on your keyboard, the *PositionIncrement()* method is invoked.

PositionIncrement()

Use this method on from an Input component to change to the next position.

PositionDecrement()

Use this method on from an Input component to change to the previous position.

ResetToInitialPosition()

This method returns the *Movable Object* in the **Mover** component to the position it started in.

MoveToElementNum(int elementNumber)

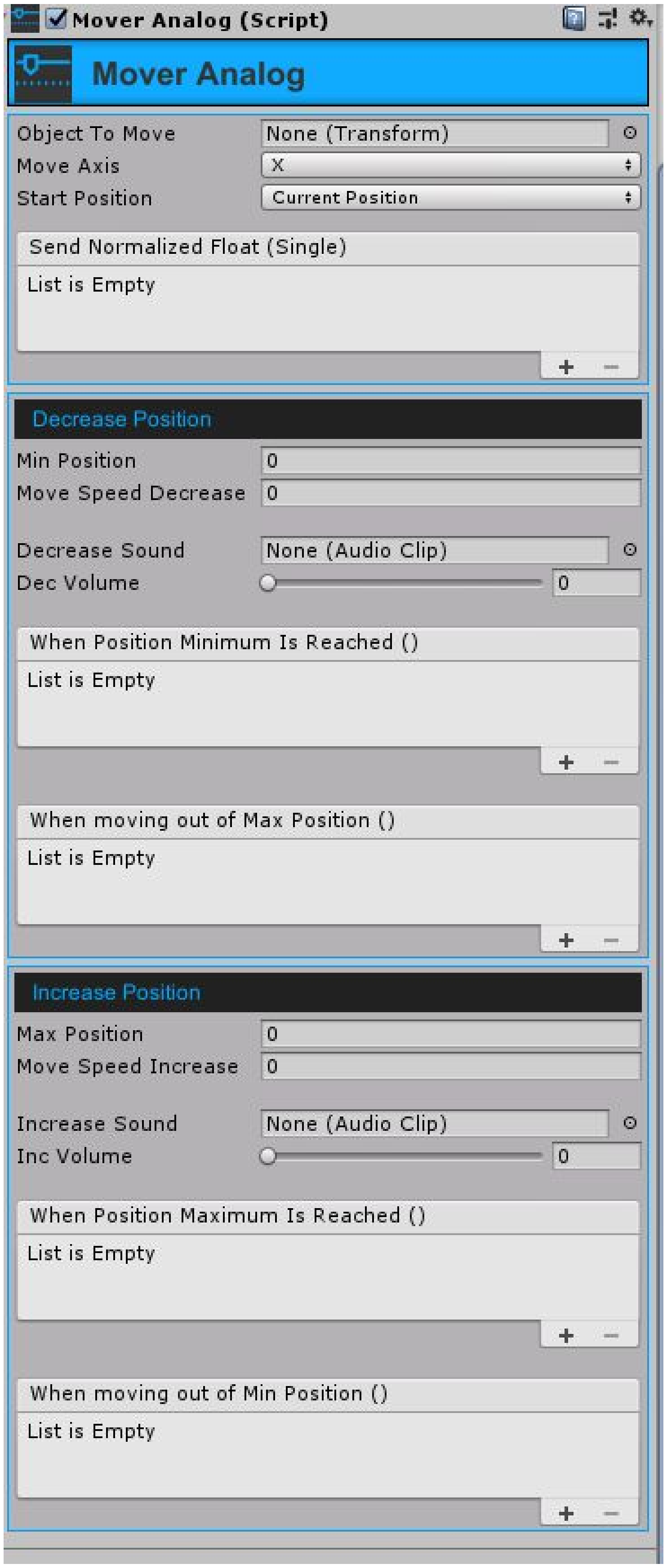
Use this method to select which position # you want the *Movable Object* to be in directly without incrementing or decrementing.

Public Properties

CurrentPositionElement

You can use this to check what state/position # the Mover is currently in.

MoverAnalog


Mover Analog (Script)

Mover Analog

Object To Move	None (Transform)
Move Axis	X
Start Position	Current Position

Send Normalized Float (Single)
List is Empty

Decrease Position

Min Position	0
Move Speed Decrease	0
Decrease Sound	None (Audio Clip)
Dec Volume	0

When Position Minimum Is Reached ()
List is Empty

When moving out of Max Position ()
List is Empty

Increase Position

Max Position	0
Move Speed Increase	0
Increase Sound	None (Audio Clip)
Inc Volume	0

When Position Maximum Is Reached ()
List is Empty

When moving out of Min Position ()
List is Empty

The **Mover Analog** component can be used to create sliders - e.g. a volume slider, or a brightness slider, or a throttle in a spacecraft or jet fighter, or any other kind of control which you want to move smoothly between two positions.

You can also create sliding sci-fi doors, stone sliding doors and moving platforms with the **Mover Analog** component.

The **Mover Analog** specifically is able to take in analog input also which allows the player to control how fast things move, or to set their absolute position between the min and max position - e.g. mapping it 1:1 with a USB Throttle HOTAS.

Object To Move

Necessity: *Required*

Accessibility: *Protected*

The Transform of the object which we will move (relative to the parent Transform).

Move Axis

Necessity: *Required*

Accessibility: *Protected*

The local axis we want to move on. If we want to move on multiple axes, then we can combine two **Mover Analog** components to the same *Object To Move* Transform.

Start Position

Necessity: *Optional*

Accessibility: *Protected*

Select where we want to our *Object To Move* Transform to start from. Current, Min or Max positions.

Send Normalized Float (Single)

Necessity: *Optional*

Accessibility: *Protected*

Each time the position is incremented or decremented, we can choose to send a Normalized Float value as an event to another method. The float value sent will be between 0 and 1. It can be useful to update another scripts that this object has moved, and how far between the min and max positions it is currently located at.

Decrease Position

These are the settings which are used every time we move the position of the *Object To Move* Transform *in a negative direction.*

Min Position

Necessity: *Required*

Accessibility: *Protected*

This determines how far we can move the *Object To Move* Transform in the negative direction before it stops.

Move Speed Decrease

Necessity: *Required*

Accessibility: *Protected*

Be sure to set this otherwise you might wonder why nothing is moving :-)
This determines the base move speed when moving in the negative direction.

Decrease Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound that is played when the *Object To Move* Transform is moved in the negative direction.

Dec Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the sound cip set above.

When Position Minimum Is Reached()

Necessity: *Optional*

Accessibility: *Protected*

When the minimum position is reached, we can choose to invoke any methods that listed in this event. E.g. If we turn down the radio volume to minimum we can choose to also switch it off completely.

When moving out of Max Position()

Necessity: *Optional*

Accessibility: *Protected*

When the position is in the maximum position and then moved from the maximum position, this event will be triggered. E.g. Great for turning off an afterburner when a fighter jet's throttle is moved out of the 100% mark.

Increase Position

These are the settings which are used every time we move the position of the *Object To Move* Transform *in a positive direction.*

Max Position

Necessity: *Required*

Accessibility: *Protected*

This determines how far we can move the *Object To Move* Transform in the positive direction before it stops.

Move Speed Increase

Necessity: *Required*

Accessibility: *Protected*

Be sure to set this otherwise you might wonder why nothing is moving :-)
This determines the base move speed when moving in the positive direction.

Increase Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound that is played when the *Object To Move* Transform is moved in the positive direction. Make this some kind of looping chain or slider or tracks sound.

Inc Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the sound cip set above.

When Position Maximum Is Reached()

Necessity: *Optional*

Accessibility: *Protected*

When the maximum position is reached, we can choose to invoke any methods that listed in this event. E.g. If we turn up the light bulb too high, it might pop.

When moving out of Min Position()

Necessity: *Optional*

Accessibility: *Protected*

When the position is in the minimum position and then moved from the minimum position, this event will be triggered. E.g. if you want an alarm to go off when something is no longer in the off position.

Public Methods

These methods should be used as part of invents in your **Input** components. E.g. Your **Analog Input** component can add the *AnalogMove()* method in it's events, so that depending on how far you move your analog stick on your controller, the speed of the **Mover Analog** *ObjectToMove* is affected directly to the how much you deflect the analog stick on your controller.

SetAbsolutePosition(float absoluteValue)

This can be used in to set the 1:1 position between your analog stick or throttle and the min/max position of the *ObjectToMove* transform

---

AnalogMove(float magnitude = 1f)

Used to increase or decrease the position of the ObjectToMove Transform, however at variable speeds depending on analog input received from a control pad.

IncreasePosition()

Increases the position with the full magnitude, so that you can use a keyboard or button instead of an analog input.

DecreasePosition()

Decreases the position with the full magnitude, so that you can use a keyboard or button instead of an analog input.

ReturnToMin()

This method makes the mover return back to the minimum position smoothly based on the decrease speed. E.g. if you want to make a door that closes automatically.

ReturnToMax()

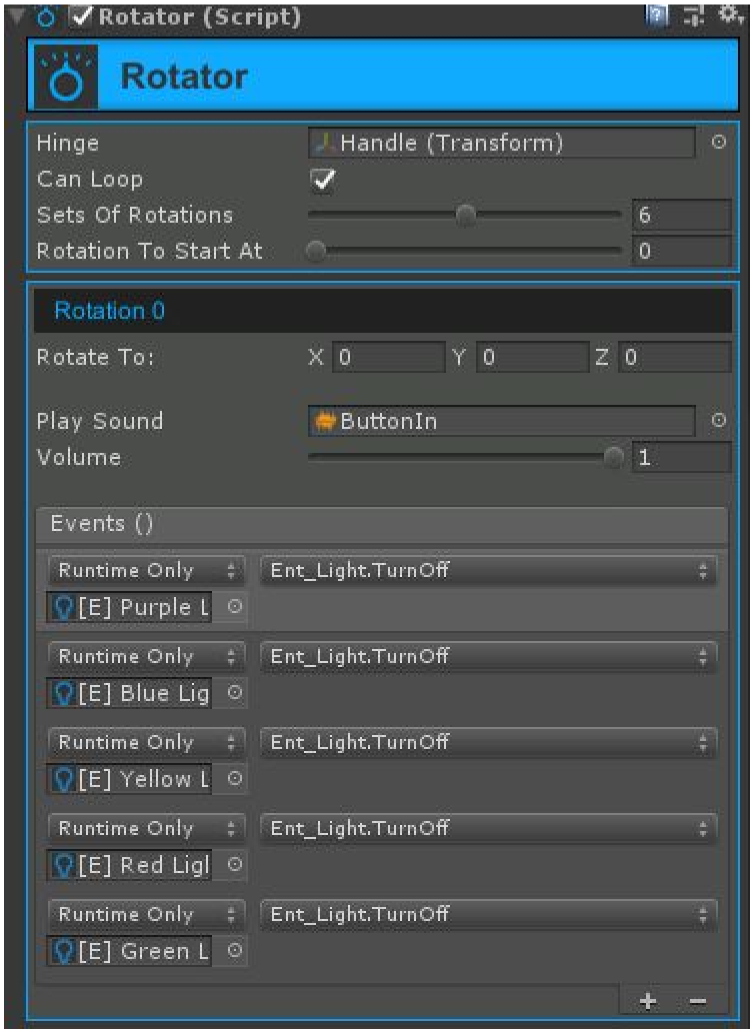
This method makes the mover return back to the maximum position smoothly based on the increase speed. E.g. if you want to make a door that opens automatically.

Public Properties

CurrentPosition

Returns a 0 to 1 value which is the scaled number of how far along the between min and max positions the *ObjectToMove* Transform is.

Rotator



Rotator (Script)

Rotator

Hinge: Handle (Transform)
Can Loop: ✓
Sets Of Rotations: 6
Rotation To Start At: 0

Rotation 0
Rotate To: X 0 Y 0 Z 0

Play Sound: ButtonIn
Volume: 1

Events ()

Runtime Only | Ent_Light.TurnOff | [E] Purple L
Runtime Only | Ent_Light.TurnOff | [E] Blue Lig
Runtime Only | Ent_Light.TurnOff | [E] Yellow L
Runtime Only | Ent_Light.TurnOff | [E] Red Ligl
Runtime Only | Ent_Light.TurnOff | [E] Green L

The **Rotator** Touchable component is used for making switches and dials that wills snap between 2 or more positions. For example a 3 position lever on a soft-serve icecream machine that switches between chocolate, off, and vanilla.

Or selecting a radio between Off, AM, FM channels. Or a car ignition

Or another example would be creating the scroll wheel on your mouse. It is snap rotates to about 20 different positions (but we could create it with just a few)

Hinge

Necessity: *Required*

Accessibility: *Protected*

The Transform which want to rotate - relative to the position of it's parent object. The hinge can be an invisible empty Transform if and then you can place the models under it with an offset, if you do not want to rotate from the middle. (e.g. if you are making a cat flap, you would want to rotate from the top area)

Can Loop

Necessity: *Optional*

Accessibility: *Protected*

When incrementing or decrementing rotations- are we allowed to go back to around to the first rotation if the last rotation has been reached (and vice versa).

Sets of Rotations

Necessity: *Required*

Accessibility: *Protected*

How many rotations we want this Touchable to have. Select between 2 and 10. Two rotations if you want to make a simple on off switch, or up to 10 rotations if you want to make a more complicated dial.

Rotation To Start At

Necessity: *Optional*

Accessibility: *Protected*

Which of our rotations should the *Hinge* field's Transform start in (note it will not run the events or play the sounds, only the Transform's rotation is changed)

Rotation

As we add rotations, each new rotation section has its own set of sounds, rotations and events to invoke as each position is reached.

Rotate To:

Necessity: *Required*

Accessibility: *Protected*

The local rotation the *Hinge* will snap rotate to when we are in this rotation *#*. It can be a completely obscure rotation if you like, on a completely different axis if you need it to be. But if you are making a dial it makes sense to rotate only one chosen axis.

Play Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound played when this rotation is reached (played by the Kinerative Manager's **Audio Source** component)

Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the above rotation sound.

Events()

Necessity: *Optional*

Accessibility: *Protected*

When this rotation reached it will invoke all of the methods in this event.

Public Methods

These methods should be used as part of invents in your **Input** components. E.g. Your **KeyCode Input** component can add the *PositionIncrement()* method in it's events, so that when you press a key on your keyboard, the *PositionIncrement()* method is invoked.

RotationIncrement()

Use this method on from an Input component to change to the next rotation point.

RotationDecrement()

Use this method on from an Input component to change to the previous rotation point.

ResetToInitialRotation()

This method returns the *Hinge* in the **Rotator** component to the rotation it started in.

MoveToElementNum(int elementNumber)

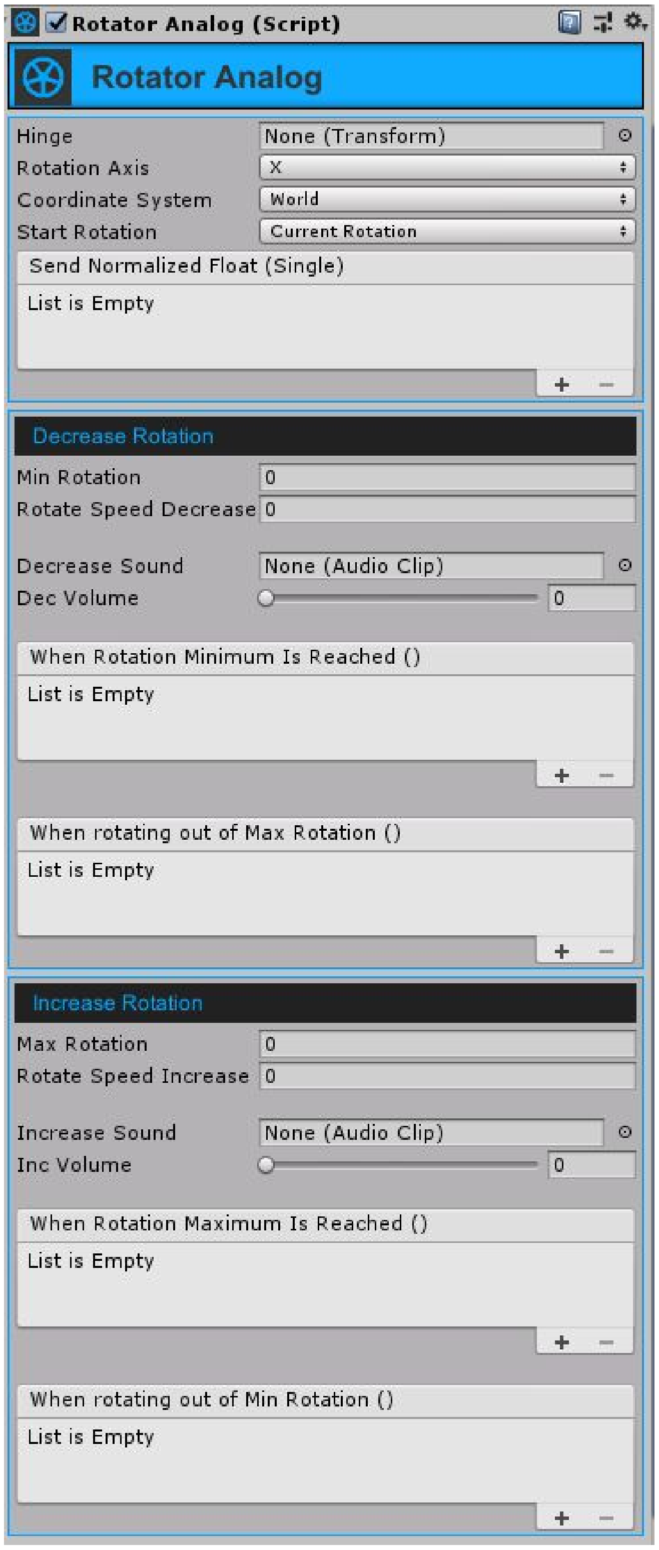
Use this method to select which rotation # you want the *Hinge* to be directly without incrementing or decrementing.

Public Properties

CurrentRotationElement

You can use this to check what state/rotation # the **Rotator** is currently in.

RotatorAnalog



Rotator Analog (Script)

Rotator Analog

Hinge	None (Transform)
Rotation Axis	X
Coordinate System	World
Start Rotation	Current Rotation

Send Normalized Float (Single)

List is Empty

Decrease Rotation

Min Rotation	0
Rotate Speed Decrease	0
Decrease Sound	None (Audio Clip)
Dec Volume	0

When Rotation Minimum Is Reached ()

List is Empty

When rotating out of Max Rotation ()

List is Empty

Increase Rotation

Max Rotation	0
Rotate Speed Increase	0
Increase Sound	None (Audio Clip)
Inc Volume	0

When Rotation Maximum Is Reached ()

List is Empty

When rotating out of Min Rotation ()

List is Empty

The **Analog Rotator** component will rotate between a min and max rotation which makes it useful for creating analog levers such as hand brakes in a car, accelerator pedals or rudder pedals as well as valves, volume dials or light dimmers.

You can also turn a rotator wheel sideways for scroll wheel type controls like on a mouse or in a fighter jet.

Tip: you can rotate something beyond 360 degrees if you want to have multiple revolutions. E.g. set your Minimum rotation to 0, and your Maximum to 720, and you will need two full rotations of the dial to reach the maximum.

Hinge

Necessity: *Required*

Accessibility: *Protected*

The Transform of the object which we will rotate.

Rotation Axis

Necessity: *Required*

Accessibility: *Protected*

The axis we want to rotate on. If we want to move on multiple axes, then we can combine two **Rotator Analog** components to the same *Hinge* Transform.

Coordinate System

Necessity: *Optional*

Accessibility: *Protected*

Changes between self and world rotation so that we can choose whether we want to rotate on the global axis or the local axis (relative to the parent object of the *Hinge* Transform.)

Start Rotation

Necessity: *Optional*

Accessibility: *Protected*

Select where we want to our *Hinge* Transform to start from. Current, Min or Max positions.

Send Normalized Float (Single)

Necessity: *Optional*

Accessibility: *Protected*

Each time the rotation is incremented or decremented, we can choose to send a Normalized Float value as an event to another method. The float value sent will be between 0 and 1. In the main example scene I've used it dim a light so that the light brightness matches the rotation of the dial perfectly (regardless of the range of the light, as it is all scaled appropriately).

Decrease Rotation

These are the settings which are used every time we move the position of the *Hinge* Transform *in a negative direction.*

Min Rotation

Necessity: *Required*

Accessibility: *Protected*

This determines how far we can rotation the *Hinge* Transform in the negative direction before it stops.

Rotate Speed Decrease

Necessity: *Required*

Accessibility: *Protected*

This determines the base rotate speed when moving in the negative direction.

Be sure to set this otherwise you might wonder why nothing is moving :-) (Although this is not affected is using the Absolute methods, since it uses the 1:1 input instead).

Decrease Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound that is played when the *Hinge* Transform is rotated in the negative direction.

Dec Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the sound cip set above.

When Rotation Minimum Is Reached()

Necessity: *Optional*

Accessibility: *Protected*

When the minimum rotation is reached, we can choose to invoke any methods that listed in this event. E.g. If we turn down the radio volume to minimum we can choose to switch it off completely.

When Rotating Out of Maximum()

Necessity: *Optional*

Accessibility: *Protected*

When the rotation is in the maximum rotation and then moved from the maximum rotation, this event will be triggered. E.g. Great for turning off an afterburner when a fighter jet's throttle is moved out of the 100% mark.

Increase Rotation

These are the settings which are used every time we move the rotation of the *Hinge* Transform *in a positive direction.*

Max Rotation

Necessity: *Required*

Accessibility: *Protected*

This determines how far we can rotate the *Hinge* Transform in the positive direction before it stops.

Rotate Speed Increase

Necessity: *Required*

Accessibility: *Protected*

Be sure to set this otherwise you might wonder why nothing is moving :-)
This determines the base rotate speed when rotating in the positive direction.

Increase Sound

Necessity: *Optional*

Accessibility: *Protected*

The sound that is played when the *Hinge* Transform is rotated in the positive direction. Make this some kind of subtle sliding sound.

Inc Volume

Necessity: *Optional*

Accessibility: *Protected*

The volume of the sound cip set above.

When Rotation Maximum Is Reached()

Necessity: *Optional*

Accessibility: *Protected*

When the maximum rotation is reached, we can choose to invoke any methods that listed in this event. E.g. If we turn up the light bulb too high, it might pop.

When Rotating Out of Minimum()

Necessity: *Optional*

Accessibility: *Protected*

When the rotation is in the minimum rotation and then rotated higher than the minimum rotation, this event will be triggered. E.g. If you want a radio’s volume knob to also turn the radio on, plus also acting as the volume knob. As the knob is rotated from the minimum rotation, the volume knob would turn the radio on, and if continued being rotated, the knob would also increase volume.

Public Methods

These methods should be used as part of invents in your **Input** components. E.g. Your **Analog Input** component can add the *AnalogRotate()* method in it's events, so that depending on how far you move your analog stick on your controller, the speed of the **Rotator Analog** *Hinge* is affected directly to the how much you deflect the analog stick on your controller.

SetAbsoluteRotation()

This can be used in to set the 1:1 rotation between your analog stick or throttle and the min/max rotation of the *Hinge* transform

---

AnalogRotation()

Used to increase or decrease the rotation of the *Hinge* Transform, however at variable speeds depending on analog input received from a control pad.

IncreaseRotation()

Increases the rotation with the full magnitude, so that you can use a keyboard or button instead of an analog input.

DecreaseRotation()

Decreases the rotation with the full magnitude, so that you can use a keyboard or button instead of an analog input.

ReturnToMin()

This method makes the *Hinge* return back to the minimum rotation smoothly based on the decrease speed. E.g. if you want to make a valve that resets when the player let's go of it.

ReturnToMax()

This method makes the *Hinge* return back to the maximum rotation smoothly based on the increase speed. E.g. if you want to make a value that turns back on when the player let's go.

Public Properties

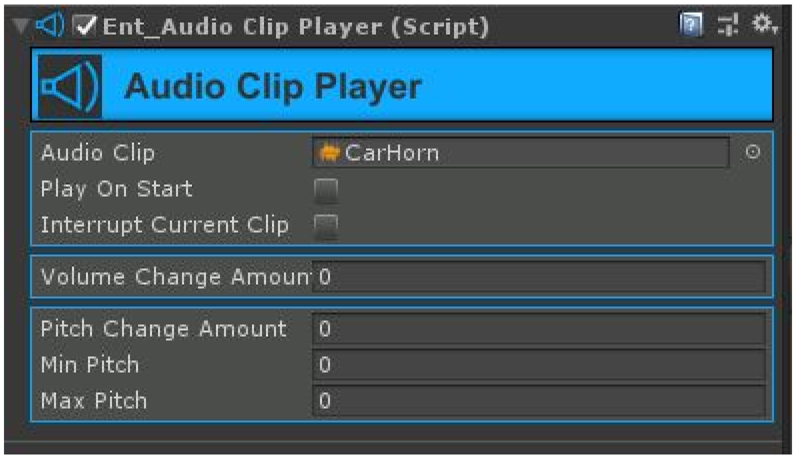
CurrentRotation

Returns a 0 to 1 value which is the scaled number of how far along the between min and max rotation the *Hinge* Transform is.

Event Entities

These event entities are mostly here as examples to help demonstrate the capabilities of KINERACTIVE. You can of course use them in your own project, however it's quite likely you will need to write specific scripts for your own game depending on what you want it to do. Anything beyond moving buttons, doors, and playing sounds and changing lighting is out of scope for this asset package. *E.g. you might want to write a spawner event entity, so that it spawns med packs when you press a button. (I can of course add this to KINERACTIVE if request)*

Audio Clip Player



Audio Clip: CarHorn
Play On Start
Interrupt Current Clip
Volume Change Amoun 0
Pitch Change Amount 0
Min Pitch 0
Max Pitch 0

This component allows a few more methods than built in Audio Source, such as changing volume by scaled amounts so that they always match the relative position/rotation of your Touchable components.

You can use it to create a car radio volume knob, or a horn or alarm button.

If you tick *Interrupt Current Clip*, then it won't wait until the clip is finished before starting again.

Public Methods

AdjustVolumeScale

Accepts a 0 to 1 value which would have been scaled by a Mover Analog or Rotator Analog component to match the full range of the rotation or movement. (Using the Send Float (single) event on those Touchable components)

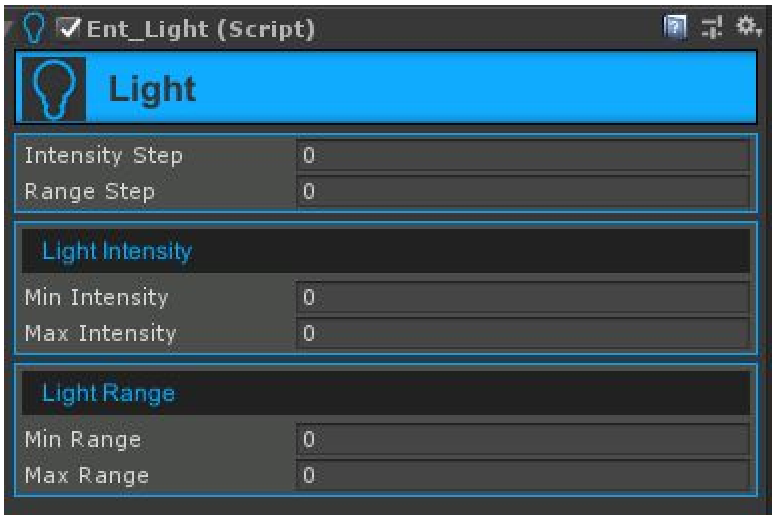
Pitch Up & Pitch Down

Changes the pitch based on the settings entered into the *Pitch* section.

Volume Up & Volume Down

Changes the volume based on the *Volume Change Amount* field.

Light Modifier


Ent_Light (Script)

Light

Intensity Step 0
Range Step 0

Light Intensity
Min Intensity 0
Max Intensity 0

Light Range
Min Range 0
Max Range 0

Allow for changing light properties and settings via the event system. E.g. Dim the lights

Public Methods

AdjustIntensityScaled()

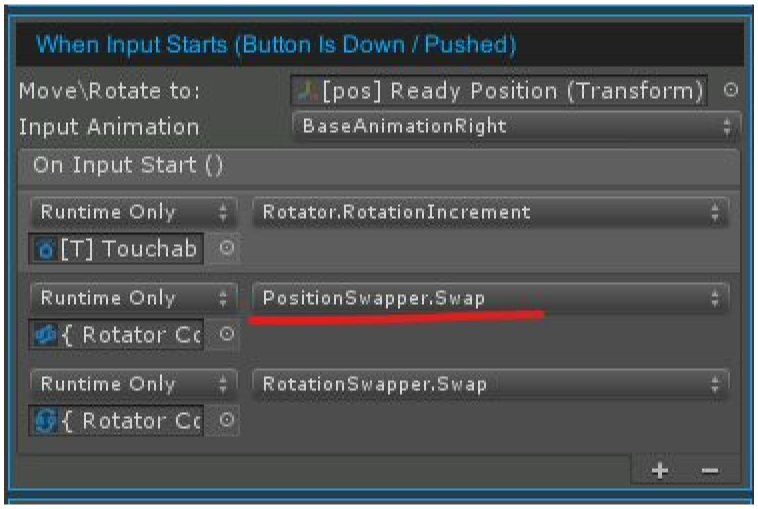
Accepts a 0 to 1 value which would have been scaled by a Mover Analog or Rotator Analog component to match the full range of the rotation or movement. (Using the Send Float (single) event on those Touchable components). This way it never goes out side of min and max parameters set on the component.

Fade In / Fade Out

This runs an IEnumerator loop to progressively fade in or fade out the light intensity

Utility Components

A number of components which either make setting up interactions easier, or making the hand / arm movements look better. You can enable/disable or run their methods from the event fields in any **Input** or **Touchable** components. (or from your own scripts).



When Input Starts (Button Is Down / Pushed)

Move\Rotate to: [pos] Ready Position (Transform)

Input Animation: BaseAnimationRight

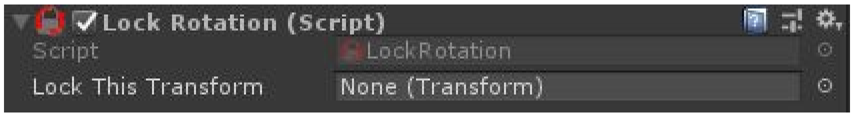
On Input Start ()

Runtime Only — Rotator.RotationIncrement — [T] Touchab

Runtime Only — PositionSwapper.Swap — { Rotator Cc

Runtime Only — RotationSwapper.Swap — { Rotator Cc

Lock Rotation



Lock Rotation (Script)

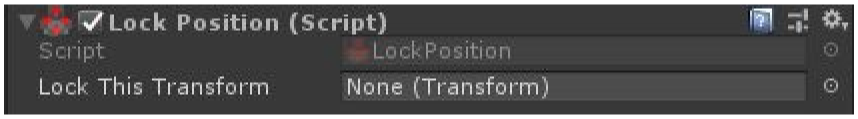
Script: LockRotation

Lock This Transform: None (Transform)

Lock the rotation of an object. Useful to make a hand follow something which flips open, such as a glove box in a car. The hand will still move with the glove box handle, however not rotate oddly at the wrist.

Just place this onto the transform you want to prevent rotation, and it will keep the rotation it starts until this component is disabled.

Lock Position



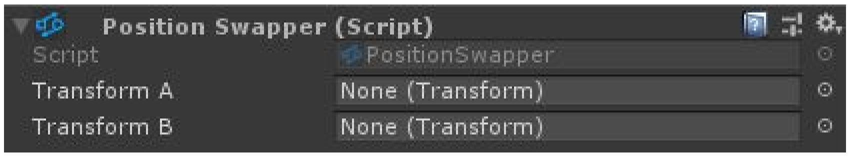
Lock Position (Script)

Script: LockPosition

Lock This Transform: None (Transform)

Lock the position of a transform. Useful if you want a player avatar's hand to match the rotation of an object, but not the positional movement. Place this script onto the Transform which you want to stop from changing position, but want to let it keep rotating.

Position Swapper

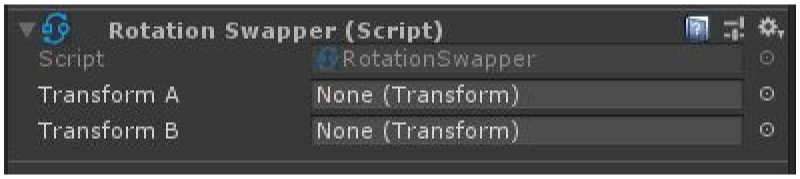


Position Swapper (Script)

Script	PositionSwapper
Transform A	None (Transform)
Transform B	None (Transform)

Switches the positions of two transforms. E.g. on a flickable switch, you can move the hand up to down (matching the motion of the switch), and then swap the two transforms so that next time your hand moves in the opposite direction, making a down to up motion. See the Toggle Button Cover in the example scene.

Rotation Swapper

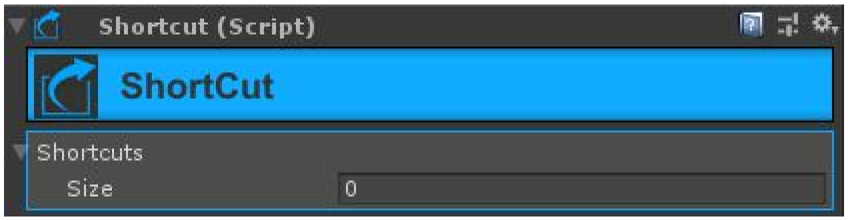


Rotation Swapper (Script)

Script	RotationSwapper
Transform A	None (Transform)
Transform B	None (Transform)

Allows you to swap the rotation settings from one transform to another.

Shortcut



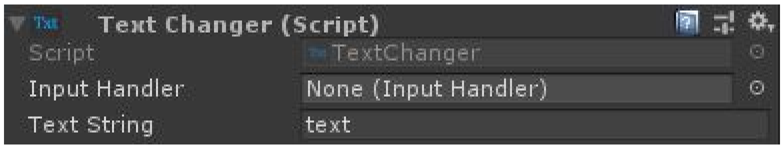
Shortcut (Script)

ShortCut

Shortcuts	
Size	0

If you have a large or deep structure of child gameobjects in the Hierarchy, you can place this component on the parent object, and add in any important child objects to the shortcut list for easy access to components hidden deep within a maze of child objects. It can be used for any project and component, not just KINERACTIVE related.

Text Changer



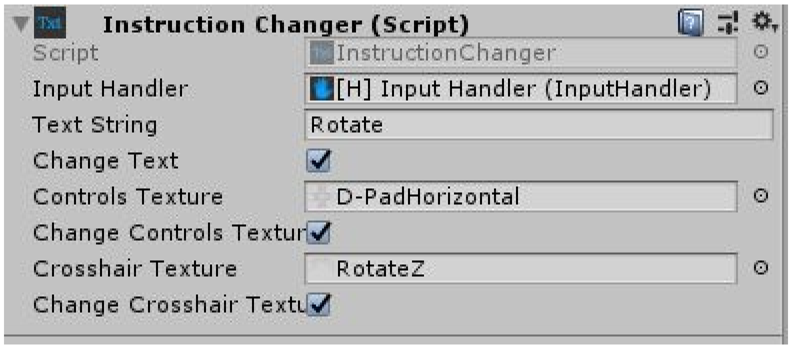
Text Changer (Script)

Script	TextChanger
Input Handler	None (Input Handler)
Text String	text

Changes text in the assigned **Input Handler** component's *Usage Instructions* field, to the text in the *Text String* field here.

E.g. Useful for making a door's instructions change from "*Open Door*" to "*Close Door*" as it opens or closes. Use this any time you want your instructions to change on an **Input Handler** component. Run the public method of *SetText()* in an event or script.

Instructions Changer

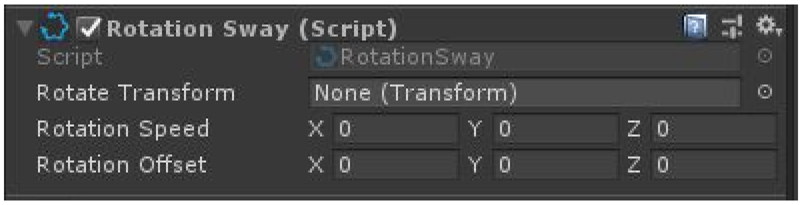


Instruction Changer (Script)

Script	InstructionChanger
Input Handler	[H] Input Handler (InputHandler)
Text String	Rotate
Change Text	✓
Controls Texture	D-PadHorizontal
Change Controls Textur	✓
Crosshair Texture	RotateZ
Change Crosshair Textu	✓

Much like the Text changer but allows for changing the crosshair, the controls popup and the text all in one go. Run the public method of *SetInstructions()* in an event or script. And it will change whatever options you have ticked in the targeted **Input Handler**.

Rotation Sway


Rotation Sway (Script)

Script RotationSway

Rotate Transform None (Transform)

Rotation Speed X 0 Y 0 Z 0

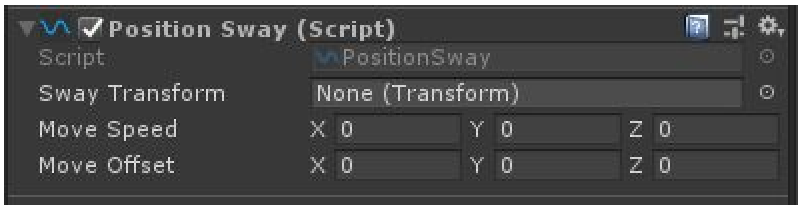
Rotation Offset X 0 Y 0 Z 0

Using sine math function, this component makes the current or assigned Transform’s rotation sway back and forth continually. It can be useful for making a hand look less static when in a “ready position” it will make minor movements to simulate hand waiting in open air.

The *Rotation Speed* sets how fast the rotation sways back and forth for each axis.

The R*otation Offset* sets how much you want each axis to sway (in degrees).

Position Sway


Position Sway (Script)

Script PositionSway

Sway Transform None (Transform)

Move Speed X 0 Y 0 Z 0

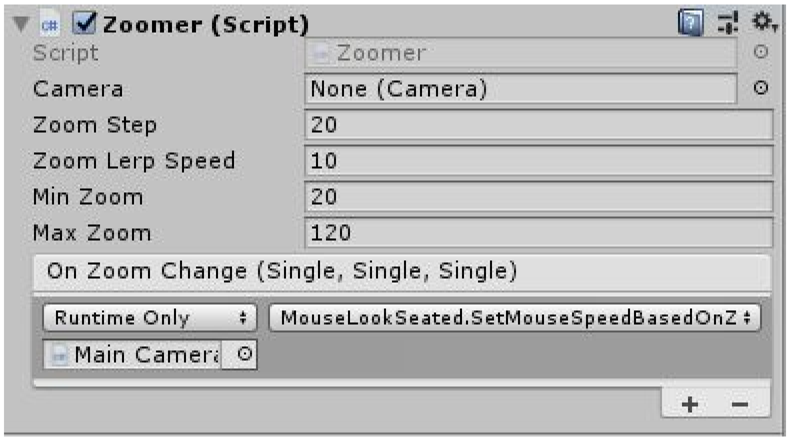
Move Offset X 0 Y 0 Z 0

Using sine math function, this component makes the current or assigned Transform’s position sway back and forth continually. It can be useful for making a hand look less static when in a “ready position” it will make minor movements to simulate hand waiting in open air. At high settings you could make it simulate vibrations e.g. if on a tractor or something like that.

The *Move Speed* sets how fast the position sways back and forth for each axis.

The Move *Offset* sets how much you want each axis to sway (in Unity units).

Zoomer


Zoomer (Script)

Script	Zoomer
Camera	None (Camera)
Zoom Step	20
Zoom Lerp Speed	10
Min Zoom	20
Max Zoom	120

On Zoom Change (Single, Single, Single)

Runtime Only | MouseLookSeated.SetMouseSpeedBasedOnZ

Main Camera

The Zoomer is a script which changes the FoV (field of view) of any given *Camera* component. I thought it might be useful if the models you are working with have a lot of details or small buttons - as many flight and space sim style games do. You also have an option for automatically lowering the mouse sensitivity at high zoom levels, which will allow fine and accurate movements of the mouse.

You can either attach a camera directly by dragging or dropping into the *Camera* field, or leave the field blank and it will automatically take the main camera (whichever camera component is marked with the *MainCamera* **Tag**)

Zoom Step is how much the FoV will change in one “step” or click of a button. E.g. if the Camera is at 60 FoV, and you press the zoom in button, it will zoom in

Zoom Lerp Speed is how quickly the FoV will change between the two steps (it’s lerped for smoothness) The lower this value is, the smoother the zoom, but the longer it will take to get to the end zoom goal.

Min Zoom and **Max Zoom** is there so that we don’t zoom in or out too far and make the game look bad

On Zoom Change is an event that is fired whenever the zoom level is changed. It also sends the min, max, and current zooms to whichever method is listening, and therefore can be used to lower the sensitivity of a *mouselook* **script** in the exact matching scale.

In the example scene, it is set up to be triggered by **Axis Input** components - mouse wheel up & down. And also a **Keycode Input** component - when holding the *Space Bar*, the we zoom in, and when we let go of the space bar it goes back to the default FoV. So there are many different ways to configure it for your game.

Public Methods

SetZoom(float zoomLevel)

Directly sets the FoV to the value set in the inspector if using the event system (or by script). This can be used for zooming to a preset level - great for if you want just a temporary zoom when holding down a button/key. Or you can have a gameplay event trigger a zoom - e.g. if you player looks at an important item, you could make it zoom in for a brief moment, or for a small cutscene. (You'd want to use Cinemachine for any major cutscenes and story moments though)

ResetZoom()

Changes the FoV back to the camera's starting FoV (it gets the value from the camera)

ZoomIn()

Increment the FoV of the camera by one Step (as set in the inspector) - controlled by the Lerp Speed setting in the inspector. E.g. use this when you want to have zoom assigned to the mouse wheel or key / axis

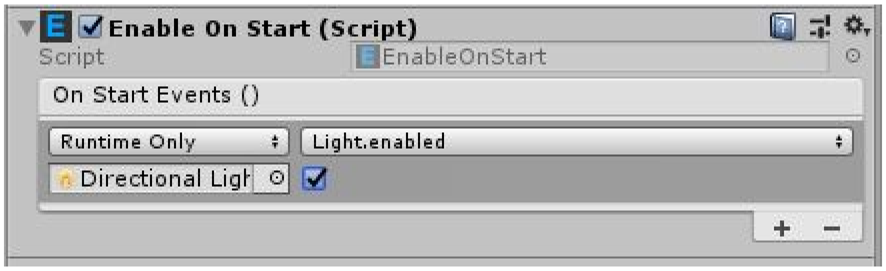
---

ZoomOut()

Decrement the FoV of the camera by one Step (as set in the inspector) - controlled by the Lerp Speed setting in the inspector. E.g. use this when you want to have zoom assigned to the mouse wheel or key / axis

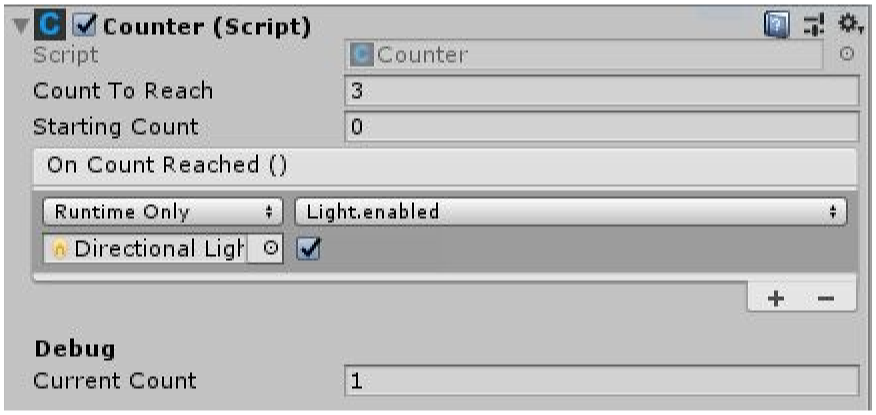
---

Enable On Start


Enable On Start (Script)
Script EnableOnStart
On Start Events ()
Runtime Only Light.enabled
Directional Ligh ✓

This is a simple script that invokes any methods which are placed into the "*Enable On Start"* event. It is used in the *1st Person Example* scene to enable the **Resting Input Handler**. Without this the **Self Input Component** would throw an error as it tries to find the **Kineractive Manager** before it has loaded into the scene. So it works as a delay to make sure everything has finished loading.

Counter


Counter (Script)
Script Counter
Count To Reach 3
Starting Count 0
On Count Reached ()
Runtime Only Light.enabled
Directional Ligh ✓
Debug
Current Count 1

This script is useful for when you want to check that several different actions have been completed. For example if your player has to pull 3 levers before a door will open, then you would set the *CountToReach* to 3, and then in each lever's event you would add the *AddToCount(1)* method. Once the *currentCount* reaches 3, the **Counter** invokes the OnCountReached event (which would open the door).

Public Methods

AddCount(int addCount)

Adds the specified amount to the current count.

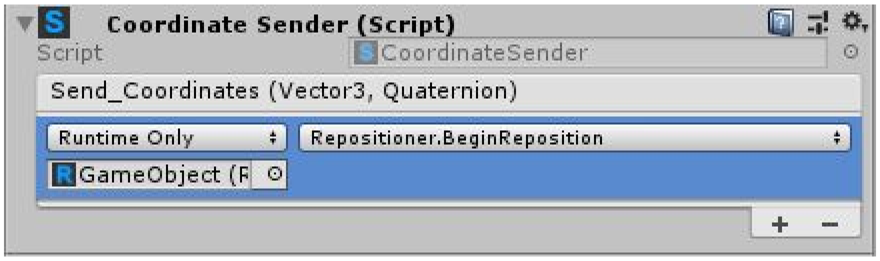
RemoveCount(int removeCount)

Takes away the specified amount from the current count.

SetCount(int setCount)

Sets the current count to the specific amount.

Coordinate Sender



Coordinate Sender (Script)
Script: CoordinateSender
Send_Coordinates (Vector3, Quaternion)
Runtime Only | Repositioner.BeginReposition
GameObject (R

The **Coordinate Sender** has one event which takes the *Position* & *Rotation* of the current **Transform** which the script is on, and sends the *Position* and *Rotation* to whichever public method has a *Vector3* and *Quaternion* in the signature. Specifically in KINERACTIVE it is used to send these values to the **Repositioner** component / script.

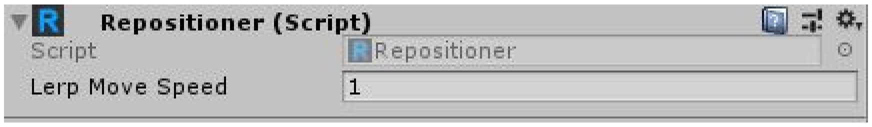
Tip: when positioning and rotating the **Transform**, make sure that the player ends up facing the **Touchable** or *interactive* components, otherwise it doesn't feel very smooth for them as they will need to look back and forth.

Public Methods

SendCoordinates()

Invokes the *Send_Coordinates* event. (You should place this method into a **Self Activated Input** component, so that when the player looks at a **Touchable** the coordinates are sent to the **Repositioner** script, and the player is repositioned to the pre-defined spot for the most smooth looking interaction with the **Touchable** object in game.

Repositioner



Repositioner (Script)

Script: Repositioner

Lerp Move Speed: 1

When the player is allowed to roam around in the world in first person mode, they can approach buttons and levers from any angle and this can create some strange situations where the hands cannot reach buttons or the hands are twisted incorrectly etc. The solution in many games is to reposition the player, and in KINERACTIVE we can do the same thing for when it's required. By repositioning the player to the ideal position and rotation, we can ensure the animation looks exactly how we designed them.

The **Repositioner** component is to be placed onto your Player's main transform. When the **Repositioner** receives *position* and *rotation* information from a **Coordinate Sender** component (or by another script invoking the *"BeginReposition()"* public method, then it will move and rotate the *Player* to the received coordinates. This way your player will be in the best spot for interacting with buttons / switches.

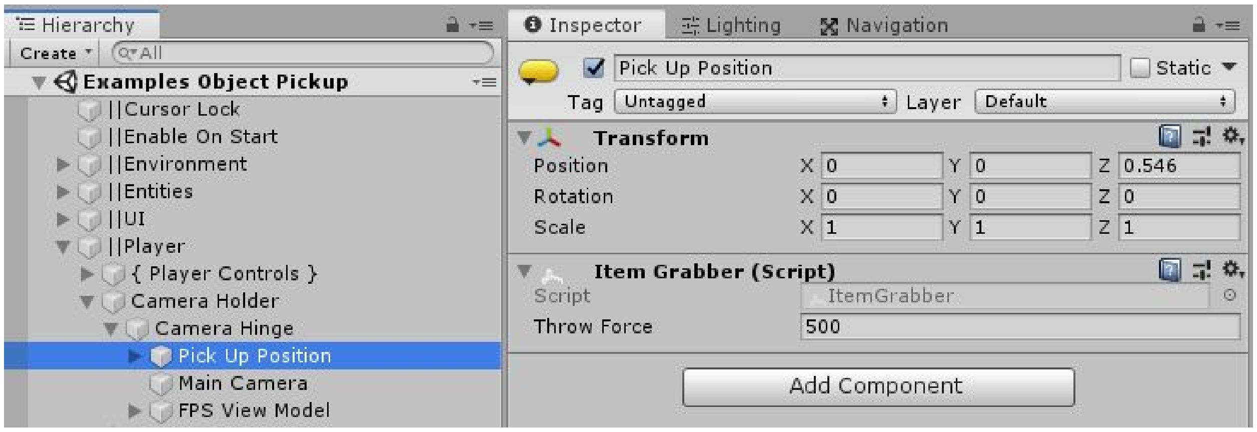
Lerp Move Speed is how quickly you want the repositioning to happen. (It's setting the repositioning time, not the rate, so no matter the distance, the travel time will always be the same (as set)).

Public Methods

BeginReposition(Vector3 newPosition, Quaternion newRotation)

Cancels any current repositioning in progress and starts the repositioning process again with the coordinates received.

ItemGrabber



Pick Up Position — Static
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0.546
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Item Grabber (Script)
Script: ItemGrabber
Throw Force: 500

The **ItemGrabber** should be placed onto a *Transform* which is located to where you want the item to go when the player character's hands are holding the item / object - ideally you should make this a child of the *camera* or *camera holder*, and position this so that the *Collider* of the picked up object is in view of the crosshair or centre of the camera, so that the **Kineractive Manager's** raycast keeps seeing the picked up object's collider. (If the object is not detected by the Kineractive Manager's raycast, it will cease the interaction)

Public Methods

GrabObject(GameObject *objectToMove*)

Execute this method via an event from an Input of the pick-up-able object or item. E.g. such as **Button Input** that is part of the object which can be picked up.

It will move the pick-up-able object to the centre of the predefined *Transform* which contains the **ItemGrabber** component/script.

It also makes any stops sets the isKinematic flaga to true, to prevent the physics simulation from interrupting while our character holds the object.

The pick-up-able object is then made a child of the ItemGrabber Transform so that it will stay in the right position as the player moves and looks.

DropObject(GameObject *objectToMove*)

Execute this method via an event from an Input of the pick-up-able object or item. E.g. such as **Button Input** that is part of the object which can be picked up. (Ideally from a release of input, so that the item is dropped when we let go of the mouse button or keyboard key)

This method gives the pick-up-able object no parent, and sets isKinematic to false. This gives the player the impression that the item is being dropped.

ThrowObject(GameObject *objectToMove*)

Execute this method via an event from an Input of the pick-up-able object or item. E.g. such as **Button Input** that is part of the object which can be picked up.

It does the same as the DropObject method, however ThrowObject also adds force to the rigidbody on the pick-up-able object to give it a throwing functionality.

SetThrowForce(float newThrowForce)

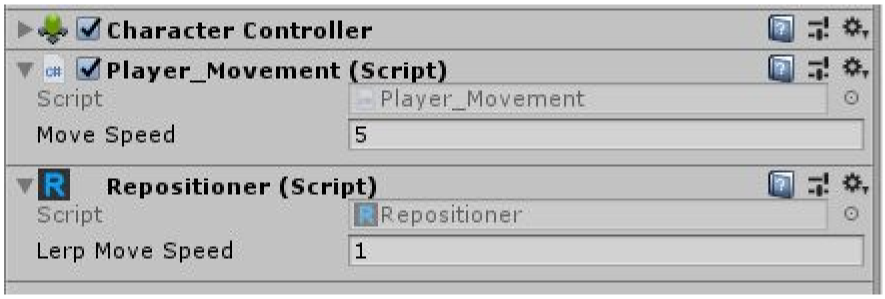
This allows you to set a higher/lower throw force during runtime by script or KINERACTIVE & Unity Events (default is 500). E.g. if your character gets strong or weaker.

First Person Example Scripts

The scripts provided in the *KINERACTIVE \ Scripts \ Controls \ FPS* are for example only, and while you're more than welcome to use or copy them in your game in any way you see fit, they'll likely need to be expanded on by you to suit your game style (you'll want to add health, inventory, jumping and crouching etc). You might also like to consider using many of the excellent Character Controllers already in the Unity Asset Store.

In any case, below are explanations of what these scripts do and how they work.

Player_Movement

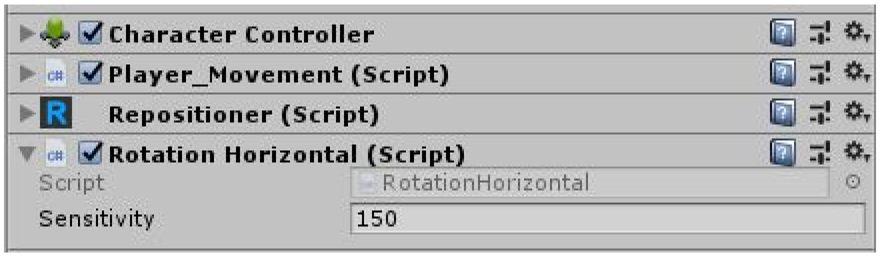

Character Controller
Player_Movement (Script)
Script: Player_Movement
Move Speed: 5
Repositioner (Script)
Script: Repositioner
Lerp Move Speed: 1

The **Player_Movement** script takes players inputs and translates them to vectors. It then passes these vectors to the Unity **Character Controller** component so that it can move the player accordingly.

Just for demonstrating more KINERACTIVE features I made this Player_Movement script gets its inputs from two KINERACTIVE **Analog Input** components, but you can change it very easily to get them from the Unity input manager instead with: *Input.GetAxis("axisName")*

The **Player_Movement** script also subscribes to events of a **Repositioner** component on the same object. This way, we will know when repositioning has started, and then ignore any player input while the player is being repositioned to ensure the player doesn't disrupt the transition.

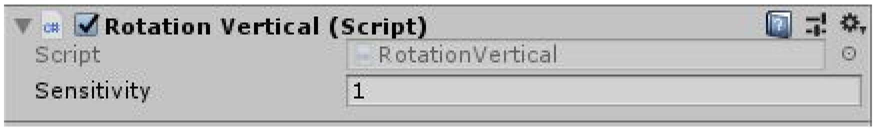
Rotation Horizontal


Character Controller
Player_Movement (Script)
Repositioner (Script)
Rotation Horizontal (Script)
Script: RotationHorizontal
Sensitivity: 150

This script takes the mouse input from the **Unity Input Manager** *"Mouse X"* axis to allow the player to look left and right. You should place it on the main player object to allow the entire player to rotate left and right as you look around.

It also waits for events from the **Repositioner** to know when to ignore mouse input during a repositioning process to prevent the player from disrupting the transition.

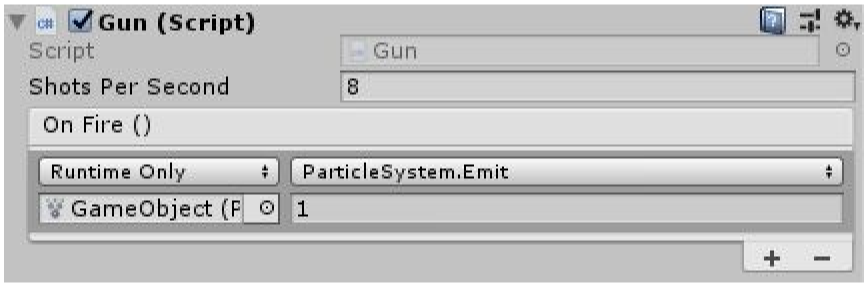
Rotation Vertical



Script: RotationVertical
Sensitivity: 1

Rotates a transform to allow the player to look up and down. This should be placed on the Camera itself or on an empty GameObject acting as a hinge (the camera would be a child of the hinge).

Gun



Script: Gun
Shots Per Second: 8
On Fire (): Runtime Only, ParticleSystem.Emit, GameObject (P, 1

This script waits for input to “pull the trigger” and when the trigger is pulled, the gun fires the specified amount of times per second. Each “shot” of the gun invokes the *OnFire()* event. (in the 1st Person example scene it just shoots out particles from Unity’s **Particle System** component, which is a child of the gun, and shoots the particles in the direction of the gun)

Weapon Holster

This script tells Unity’s Animator component to point the gun model upwards so that it doesn’t get in the way when the player is using buttons or switches. And then moves the gun back down again into aiming position.