KINERACTIVE 1.11

Manual

Introduction

First of all, thank you for purchasing this package! It let's me know that my hard work is appreciated. If you find this package useful, I hope you will find the time to leave a quick review on the Unity Asset Store. This will give other developers reassurance that they are investing into a package which will help them achieve their goals. These additional sales allow me to dedicate more of my time on improving KINERACTIVE to make it an even better product for you, which means all of us developers benefit - all thanks to your review :-)

Working on this package has made me realise that I need to look at things with a certain mindset to set up interactions in optimal ways - for example sometimes it requires a bit of creativity with the Unity Hierarchy to make a switch or button work the way I want . It has taken me a week or so to get into this mindset, and I have tried to demonstrate this by creating example scenes of the best possible ways to use KINERACTIVE - I've tried to account for as many different scenarios as I can think of to make sure it is the all encompassing interaction tool I want it to be.

I would also love to see examples of KINERACTIVE being used in your projects! Keep in touch on Unity Connect :-)

-Tom

Support

If you have any issues or concerns, or just want some advice, please contact CleanShirtLabs@gmail.com and then we can set a mutual time where we can communicate directly to get you working as soon as possible. This email is checked on a daily basis, so please allow up to 24hrs to receive a response. I will help you out as best as I can - Perhaps there is a creative solution, or perhaps I need to add more code or documentation. Either way, please feel free to reach out to me.

A lot has changed between 0.9 and 1.0, so I understand if that is annoying, but I figured it would be best to do it all now rather than bit by bit - I don't plan on any major changes like this in the future, now that I have set out the framework, so it is safe to go wild with all features knowing that future updates won't break your projects.

Roadmap

With most software, there's always more that can be added to make the software or code base even better, easier to use and more useful. I plan on adding features regularly in order to expand the use cases for KINERACTIVE and to have the animations more natural looking. I believe the best way to add features is to add them in sets of complete and fully functional components (instead of heaps of partially finished features), so this will be my philosophy moving forward.

The roadmap below isn't set in stone, and will largely depend on how *you* the customer ends up utilizing this package. I will attempt to tailor the release of features to meet the needs of my customers. I'm hesitant to provide time frames but I expect the feature updates to be fairly regular, and bug fixes very quickly.

As far as I know, nothing else like KINERACTIVE exists, so we have a chance to mold and shape the KINERACTIVE package together into a unique asset and tool.

1.0 Base Release

Major overhaul of the Early Access release to allow for continued updating (without affecting existing projects in any major ways post 1.0)

1.01 & 1.02 Bug fixes

Contains several much needed bug fixes.

1.1 FPS Focused

There is an example scene with the basics of a first person controller, a working gun and interactions. The example scene demonstrates how all of the new components fit together.

1.2 VR Focused

VR brings some unique challenges which I haven't addressed yet, and as with 1.1 I'd like VR to work out of the box as well

1.3 More Features

Ease of use improvements, as well as solutions to unique cases and scenarios, and little details that make the hand animations more customisable, flexible and realistic.

How it works

While there is a lot of code and a lot of time spent into perfecting the functionality of KINERACTIVE, the core concept of it is actually quite simple - we're using just two base mechanics:

1) The built in IK system makes your character's hands follow a blank Transform. KINERACTIVE then moves this blank Transform around as needed. (We're animating the hands by moving a blank Transform around, on which the hands are attached to - like a puppet on a string. KINERACTIVE is used to program the string movements)

2) A raycast is used to check which objects can be interacted with, and when one is detected, KINERACTIVE activates, and starts the movement. (we can then do additional movements based on the player's input from controllers/keyboard/mouse)

That's basically it. The above two base mechanics are controlled through 6 types of components which are part of KINERACTIVE, all working together. The 6 types, are:

1. **Kineractive Manager**

   Turns on *Input Handlers* when you look at an interactive object. (and off again if you look away)

2. **IK Control Script**

   Makes the character's hands follow the blank Transform. Just needs to be in the scene.

3. **Input Handlers**

   The *Input Handler* makes each Input component check for input each frame.

4. **Inputs**

   When the Input handler is active, any number of these components listen for the player's input from either controllers/keyboards/mice

5. **Touchables**

   These components are the code / logic used to make things move - such as a switch being flipped. This "switch flip" code would be run from an event in any associated Player Input component.

6. **Entities / Events**

   These are quite game specific, but two things most games have sounds and lights, so I've provided two components as examples to help demonstrate how things should be set up in KINERACTIVE's framework.

7. **Utility**

   These components help modify existing behaviour of other components to help the procedural animations come to life and look more natural.

See the Component Reference document for detailed descriptions of what each component does and what each field is used for.

Getting Started

Installing KINERACTIVE

First install the prerequisite models and animations Unity package located here:

https://drive.google.com/file/d/1Qa0X3tWX8QaYNl0ADWMbincgoPavCqrA/view?usp=sharing

(License restrictions do not permit me to keep these inside store's KINERACTIVE package, however they are free for both personal and commercial use - check the MIXAMO license details for more info)

After that is imported - install the KINERACTIVE package from the Unity Asset Store.

Preliminary Check

Open up any one of the example scenes located in the KINERACTIVE \ Scenes folder.
Look around with the mouse and try to interact with a few of the touchables. There should be no errors. If there are any, see if you can identify what the cause might be - perhaps there is a conflict between KINERACTIVE and another project in your package, or perhaps KINERACTIVE didn't import correctly etc. In any case, we need to sort out the issue before we can continue. Contact support if you need help :-)

When you first read through the section below be careful not to get confused between the various button components:

- **Button Input** component (listens for real world button presses from the Unity Input Manager, such as Fire 1, Fire 2, Submit etc)
- **Button Touchable** component (the logic for the button inside the game world)
- The **Button Model**, the 3D game model - visual representation of the in game button.
- Mouse **Buttons** - the things that make that cool clicky sound on your mouse in real life

Also See the Player Inputs and Player Anims section at the end of this document if you plan on changing the names of the default *Unity Input Manager* inputs.

Setting Up Your First Scene

To get familiar with how KINERACTIVE works, it's best to start a blank scene, perhaps even in a blank project, and follow along with the steps below. Once you are happy with how it all fits together, then you'll be able to incorporate KINERACTIVE into your own project with confidence. You can follow the steps below in your own project too, but essentially just swapping out the KINERACTIVE models, animators and UI with your own.

If you get stuck, you can use the **Getting_Started_1** scene as a reference. It was made to match the steps below exactly. (Located in Assets \ KINERACTIVE \ Scenes)

Set up the KINERACTIVE Manager component

To set up the Kineractive Manager component, we need to add this component into the scene, put in all of the prerequisite objects, and then link them to Kineractive Manager component.

The prerequisites are: the Player Model, an Animator, Animator Controller, an IK script, UI canvas with UI Text & UI Image, Audio Source, player inputs file, player animations file, a few empty "helper" transforms, and of course the Camera.

Add the Kineractive Manager to the scene

1. Start a new blank scene
2. Create a new empty GameObject in the Hierarchy, and rename it to **Kineractive Manager**
3. Press the *Add Component* button in the *Inspector* and find the *Kineractive Manager* component under KINERACTIVE > Prerequisite . Or drag and drop it from the Assets \ KINERACTIVE \ Scripts \ Prerequisites folder in the project window.
4. Reset the position/rotation/scale in using the Cog icon in the Transform of this GameObject.

Add The Player Model

5. Create an empty GameObject in the Hierarchy to act as a folder for your player, name it **Player**
6. Reset the position/rotation/scale of the **Player** Transform in the Inspector

7. Create an empty GameObject under the **Player** GameObject, and rename it to **Model**
8. Reset the position/rotation/scale of the **Model** Transform in the Inspector

9. Go into Assets\ KINERACTIVE \ Models Project folder and drag the *Swat Model T* into the **Model** GameObject in it the Hierarchy
10. Reset the position/rotation/scale of the **Model** Transform in the Inspector

11. Right click on the model's GameObject (named **Swat Model T**) and select *Unpack Prefab Completely*.
12. In the Inspector, remove the *Animator* component from the **Swat Model T** GameObject

Add the Animator

13. Click on the **Model** GameObject you created earlier, and add a new *Animator* component. (By placing the animator onto the model gameobject, instead of the model itself, it allows for easy model swapping later on)
14. In the Avatar field, put in the **Swat Model TAvatar** avatar (click the and select from the popup window)

Add the IK Control script

15. In the *Controller* field of the *Animator*, add the **SittingPlayer** *Animator Controller* from Assets \ KINERACTIVE \ Animator Controller folder in the Project window.
16. Add the *IK Control* script just below the *Animator Component* on the **Model** GameObject (located in Assets\ KINERACTIVE \ Scripts \ IK)

Add The Camera

17. Place the **Main Camera** GameObject into the **Player** GameObject (as a child)
18. Reset the position/rotation/scale of the **Main Camera** Transform in the Inspector

Add an AudioSource component

19. Create a new empty GameObject in the Hierarchy. Rename to **Kineractive Audio Source**
20. Add an *AudioSource* component;

21. Drag this GameObject in the **Kineractive Manager** GameObject as a child.

Add IK Helpers

22. Create 4 blank GameObjects. Name them **LeftHandRestPosition, RIghtHandRestPosition, LeftHandIKHelper, RightHandIKHelper**

    These will help us make the hands move, as well as help return them to resting position

23. Drag these GameObjects into the **Kineractive Manager** GameObject as a children.
24. Rest all of their position/rotation/scales

Create the UI

25. In the Hierarchy, click and from the menu select **UI > Canvas** and rename it to **UI**
26. Reset the Transform's position/rotation/scale
27. Right Click on the **UI** GameObject and from the menu select <u>UI > Raw Image</u>
28. Rename the **Raw Image** GameObject to **Crosshair** in the Hierarchy
29. With **Crosshair** GameObject selected:
    a. In the Raw Image(Script) component, click the *Texture* field's widget, and from the pop up window, select a suitable crosshair texture (such as the one called **CrossStrippedInner128**)
    b. Change the Color to something vibrant, such as red.
    c. Untick *Raycast Target*

30. Right Click on the **UI** GameObject again, and from the menu select <u>Create Empty</u>
31. Rename this GameObject to **Instructions**

32. Right click on the **Instructions** GameObject and from the menu select <u>UI > Image</u>
33. Rename the **Image** to **Text Background**
34. With Background GameObject selected:
    a. set the Image (Script) component to disabled (untick)
    b. select the color to *black*, with *127* alpha
    c. Untick *Raycast Target*
    d. Set the RectTransform to
        i. Width*: 450*

        ii. Height*: 60*
        iii. Pos Y*: -150*

35. Right click on the **Text Background**GameObject and select from the menu UI > Raw Image
36. Rename the **Raw Image** to **Controls Icon**
37. With **Controls Icon** GameObject selected:
    a. *Set the RectTransform to*
        i. *Left Middle*
        ii. Width*: 50*
        iii. Height*: 50*
        iv. Pos X*: 10*
        v. Untick: *Raycast Target*

38. Right click on the **Text Background**GameObject and select from the menu UI > Text
39. Rename the **Text** to **Interaction Text**
40. With **Interaction Text** GameObject selected:
    a. *Set the RectTransform to*
        i. *Stretch*
    b. Set the Text (Script) component to
    c. Font: *Arial*
    d. Font Style: *Bold*
    e. Font Size: *26*
    f. Alignment*: Middle / centered*
    g. Set the Color to: *white*
    h. Untick: *Raycast Target*

Join Everything to the Kinerative Manager

41. Click on the **Kinerative Manager** GameObject in the Hierarchy. We can now fill in all of the fields with the components and objects we've added to the scene.

42. In the *Detection* section of the **Kineractive Manager**
    a. Set how many times we want the Raycast to happen - 10 per second is fine for a responsive interaction.

    b. Set how far we want the *Raycast Distance* to shoot the ray - 1 Unity unit (one metre) is a good value. Ideally have it roughly the length of your character's arm. Set as required.

    c. The *Ray Origin* should be set to your Main Camera (that way wherever the camera looks, the ray will fire in the same direction, therefore whatever we look at, we can interact with - provided it's within range)

    d. Set the *Layer Mask* to **Everything** . This will make the Kineractive Manager check every object in your scene.

43. *Hands* Section
    a. Drag and drop the Left/Right Hand **Rest** and **IK Helper** GameObjects you created in earlier into the matching the fields in the *Hands Transforms* section in *Kineractive Manager* component

    b. Into the *Hand Animator* field, drag and drop the **Model** GameObject from the Hierarchy which you made earlier (it was located as a child in the **Player** GameObject). This will automatically select the Animator component for us and place that into the field.

    c. Into the Player Anims field, drag and drop the **defaultAnims** ScriptableObject which is located in <u>Assets \ KINERACTIVE \ _Config \ Anims</u> in the Project window. (this file is a list of which animations which the *Kineractive Manager* is allowed to play)

   d. *Feet* section can be left empty

44. *UI* section
    a. Drag and drop all of the **UI** GameObjects we made earlier into the appropriate fields (the names match the fields) from the Hierarchy

45. *Audio* section
    a. Drag in the **Kineractive Audio Source** GameObject from the Hierarchy into the Audio Source field.

46. *Player Inputs* section
    a. Into the Player Anims field, drag and drop the **defaultInputs** ScriptableObject which is located in <u>Assets \ KINERACTIVE \ _Config \ Inputs</u> in the Project window. This file tells the Kineractive Manager which Unity Input Manager Buttons and Axes can be used)

47. Press ***<u>Play!</u>***
48. There should be no errors, and the model should be in a seated position (with the hands probably going somewhere strange - this is good, it means they are follow our IK Helper Transforms). While in "Play Mode" you can drag around the RightHandResPosition and LeftHandRestPosition in the Scene view, and they character's hands will follow. This means everything is working as expected.
49. If there are errors, then please go through and check that the Kineractive Manager component is set up correctly. If you get stuck, compare against the provided scene, or contact me and I'll be happy to help you out.

OK, so that was a lot of steps, but the good news is, we can prefab all of this for next time. It's just a good idea to go through it all at least once, to get an idea of how it's all connected. The set up will be more or less the same for every one of your projects.

Adding Mouse Look & Camera Setup

Now that we have the prerequisites set up, lets put in the ability to look around as if we are seated in a car or cockpit. This will vary a little for each model, but essentially we are hiding the head of the model, and putting the camera in its place so that we can have the viewpoint through the eyes of the character. This section corresponds to the scene named **Getting_Started_2**

1. Use the Hierarchy search to find the object called **swat:Head**, and set the scale to *0,0,0.* So that the character's head completely disappears.

2. Find the **Main Camera** GameObject in your scene (likely tucked away as a child under the **Player** GameObject).
3. Set the *Clipping Planes Near* to 0.01 on the Camera component.
4. Add the *Mouse Look Seated (Script)* component. Either by dragging it in from the Project window in Assets \ KINERACTIVE \ Scripts \ Control or by pressing the Add Component button in the Inspector - it is located in KINERACTIVE > Control
5. Press ***play,*** and move your mouse around to look around.
6. (while still in play mode) In all likelihood, the camera is placed near the player's feet, which is no good, so adjust the **Main Camera** position until it feels right - just above the neck of the sitting player. Take your time and get it just right.
7. Use the cog drop down menu on the **Main Camera** Transform component to *Copy Component*
8. **Stop** the Play mode.
9. Now use the cog again on the Main Camera, and select *Paste Component Values* this will move the camera to the optimal position which found during play mode. Now everytime we hit Play, it will always be in the optimal spot (even though it looks strange int the editor, since the animations are not playing).

10. Now we can use the same process to set the positions of our LeftHandRestPosition and RightHandRestPosition GameObjects.
11. Press Play, move the **LeftHandRestPosition** GameObjectto somewhere around the thigh or knee area of the character model.
12. Copy the Transform component, stop play mode, and paste the Component values back into the Transform.
13. Repeat this for **RightHandRestPosition** so that both of the character's arms are in his lap.

Once again, we can prefab this character set if want to. Now the character setup is finished, and we are ready to touch some interactive items!

Creating Our First Button

A completed setup of the below instructions can be viewed in **Getting_Started_3**.

Ideally you should use your own button or switch models, but I have provided some made out of Unity primitives for demonstration purposes.

Add the Button Model

1. Create a new empty GameObject in the Hierarchy, and rename to **{ Button }**
2. Create a child GameObject under **{ Button }** and rename it to **[M] Model**
3. In Assets \ KINERACTIVE \ Prefabs\ Touchable Models are several buttons and switches that can be used for prototyping. Drag the **Button Box** model prefab into the **[M] Model** GameObject in the Hierarchy.
4. Right Click on **Button Box** and select *Unpack Prefab Completely*
5. Now position the **{ Button }** GameObject so that everything is within arms reach of the character. (make sure you have the **{ Button }** object selected, not the model or children of model, it's important to move the entire structure from the highest parent object)

Add The Input Handler

6. Create a new empty child GameObject under **{ Button }** and rename it to **[H] Input Handler**
7. Add a *Box Collider* component to **[H] Input Handler**
8. Resize the Box Collider component to roughly match the dimensions of the model
9. Tick *Is Trigger*

   (we need this trigger or collider, either works, so that the Kineractive Manager raycast has something to hit)

10. Now add the *Input Handler* component to the **[H] Input Handler** either from the Project window in Asset \ KINERACTIVE \ Scripts \ Input Handers or press the Add Component button, and find it under KINERACTIVE > Handler
11. Let's fill out the fields in the Input Handler component.

a. Usage Instructions type in: *Left Click to Press*
b. Controls Icon Texture: press the ▯ and select the **MouseLeftClick** image
c. Crosshair Texture: press the ▯ and select the **Dot128** image

These images and text will appear when our player interacts with this button (they will be passed from the Input Handler component to the UI system we set up earlier)

Move The Hand

In this part of the tutorial, we finally get to see the hands move on their own! We will set up the button so that the character's left hand moves near the button, and back to the character's knee when we look at the button. We do this by setting the position of the hand, and then telling the *Self Activated Input* component that's where we want it to move to.

A completed setup of the below instructions can be viewed in **Getting_Started_3**.

Add our Ready Hand Position

12. Create a blank child GameObject under **Button Box**, and name it **[pos] Ready Position** and position it closes to the button model. This will be where our hand moves to, when we simply look at the button model. We'll need to adjust this later, so just estimate on the location for now.

Add the Self Activated Input component

13. Create a child GameObject under **[H] Input Handler** and rename it **[I] Self Activate Input**
14. Now add the *Self Activated Input* component to the **[I] Self Activate Input** GameObject either from <u>Asset \ KINERACTIVE \ Scripts \ Inputs</u> in the Project window, or from the <u>KINERACTIVE > Inputs</u> Add Component button menu

15. From the Hierarchy select the **[I] Self Activate Input** GameObject, and while selected, drag the **[pos] Ready Position** GameObject into the *Move\Rotate to:* field in the **[I] Self Activate Input** component. (under the *When Self Activated Input Enables* section of the component)

16. In the Hierarchy, select the **[H] Input Handler** GameObject, and with this selected, drag the **[I] Self Activate Input** GameObject on top of the *Kineractive Inputs* field in the *Input Handler* component to add it to the list of Kineractive Inputs in the Input Handler. You should now have a Kineractive Inputs with a size of 1, and Element 0 should reference the **[I] Self Activate Input**

17. Press ***Play*** aim the crosshair at the button, and if everyone went to plan, we should see the left hand of our character move towards the button, and the on-screen UI should also display the icons and instructions we entered.
18. It's quite probable that our hand position is not quite right, and the button is self might be too far away etc. So take the time to move them around and get things looking right - don't forget to also rotate the **[pos] Ready Position** GameObject, because that is part of making things look natural . Use the play mode to move them, and copy their Transform, and paste the values back in again after exiting play mode. Once you have this one button setup, it's a lot easier to set additional buttons, since we get a feel for the dimensions of our character and so forth.
19. We can also go into our **UI** GameObject, and filter down into the **Controls Icon**, where we can disable the *Image* component, as well as the *Text* component in the **Interaction Text** GameObject. This way these instructions only popup when we look at our button.

Pressing The Button

Now that we can look at the button, and our hand reaches toward it, let's set it up so that we can actually press it. (The *Self Activated Input* is actually optional, however it makes things look much better, otherwise our character can appear to slap the button, which at least in this case we don't want).

Below are the instructions which will make our hand move from the *Ready Position* to the *Pressed Position*, when we click the *Left Mouse Button*, and back again to the *Ready Position* when we release the *Left Mouse Button.* (see **Getting_Started_4** scene for completed steps)

Add the Pressed Position

1. Duplicate the **[pos] Ready Position** GameObject, and rename it to **[pos] Pressed Position**, move it slightly forward in the Z axis (at least .1 units), just enough so that it's not the same as Ready Position, but still near the button. We will adjust it some more later.

Add the Button Input component

2. Select the **[H] Input Handler** GameObject, and create a new child empty GameObject, and rename it to **[I] Button Input**.
3. Select **[I] Button Input** and add the *Button Input* component from the <u>Assets \ KINERACTIVE \ Scripts \ Inputs</u> Project window or by pressing Add Component, and finding it under <u>KINERACTIVE > Inputs</u> .
4. In the *Button Input* component
    a. Take note of which real world button we will press to initiate the in game button press - by default it should be FIRE 1, which in Unity is either the Left Mouse Button, or the Left CTRL key.
    b. Then in the section labelled *When Button Is Pushed Down* we need to input a Transform. Drag and drop the **[pos] Pressed Position** into the *Move\Rotato to* field.
    c. Also in this section, choose the *Input Animation* from the drop down field (by default is set to *BaseAnimationLeft*, which just plays whatever animation is on the entire body of the character. But we want our character to use their thumb, so select the animation called *ThumbLeft*.
    d. In the section labelled *When Button Is Released* we need to input where our hand will return to, this can be anywhere but in this case, let's make the hand return back to the "ready position" . Drag and drop the **[pos] Ready Position** into the *Return Position* field.

5. In the Hierarchy, select the **[H] Input Handler** GameObject, and with this selected, drag the **[I] Button Input** GameObject on top of the *Kineractive Inputs* field in the *Input Handler* component to add it to the list of Kineractive Inputs in the Input Handler. You should now have a Kineractive Inputs with a size of 2, and Element 1 should reference the **[I] Button**

Input and Element 0 will show the **[I] Self Activate Input**. (the actual order doesn't matter in this case, as they are both evaluated in the same frame, as long as both are in there, it's all fine).

6. Press ***Play!*** We can now look at the button, and the hand will 'get ready'. If we click our *Left Mouse Button* or press *Left CTRL*, we'll have the thumb move toward the button as if we our character was using his thumb to press it. If it looks strange, then simply adjust the **[pos] Pressed Position** GameObject's position and rotation until you are happy with the result.

7. To make adjusting the *Ready* and *Pressed Position* Transforms easier next time, you can use the **Left** or **Right Holo Helpler Hand** prefabs. This helps you get the hand's into more or less the correct position on your first try. Just delete the holographic hand model once you're done. (I just wanted to show the more manual process first)

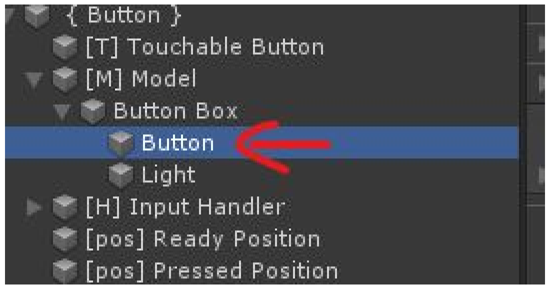
Animate the Button

Now that we can press the button - let's make the button move in and out as we press it so for more realism and to provide feedback to the player. We can write our own script for this, or we can use one of the *Touchchable* components. In this case we want the *Button Touchable* component. (See the **Getting_Started_5** scene for a completed example of this section)

Add the Touchable component

1. Create a new empty GameObject as a child under the **{ Button }** GameObject.
2. Rename the new GameObject to **[T] Touchable Button**
3. Select **[T] Touchable Button** and add the *ButtonTouchable* component located in Assets \ KINERACTIVE \ Touchables in the Project tab, or click the *Add Component* button and select KINERACTIVE > Touchables

4. With **[T] Touchable Button** selected, we can see all of the options that need to be filled out in the *Button (Touchable)* component
    a. In the first field labeled *The Button:* we want to add in the 3D model of the button that will move. In this case it's our **[M] Model \ Button Box \ Button** GameObject.



If you select this GameObject you move the Y position to test out how it moves in and out of our Button Box. Add the indicated **Button** Game Object to the first field in the *Button (Touchable)* component which is on the **[T] Touchable Button** GameObject. Either Drag and drop it, or select the little Unity circle to locate it in the scene.

b. Next in the *Button (Touchable)* component, go to the section called *Button Pressed.* This is where we enter what happens to his button when it is pressed. We want to change the pressed position, and make a click sound too.
   i. In the *Pressed Position* Vector 3, change the *Y* value to *0.2*
   ii. Set the *Pressed Clip* to *ButtonPressed.wav l*ocated in Assets \ KINERACTIVE \ Sounds

c. Under the Section labelled *Button Out*
   i. Set the *Out Position* to *X*:0 *Y*:0.4 *Z*:0
   ii. Set the *Out Clip*: *ButtonOut.wav* located in Assets \ KINERACTIVE \ Sounds

Connect the Button (Touchable) to Button Input

1. Find the **[I] Button Input** GameObject and select it.

   (it should be a child of [H] Input Handler)

2. In the section of the *Button Input* component called *When Button Is Pushed Down* click on the little +plus icon to add an event.
3. Drag and drop the **[T] Touchable Button** GameObject into the empty field of this event.
4. From this event's drop down list, select ButtonTouchable > PressButton()

5. In the section of the *Button Input* component called *When Button Is Released* click on the little +plus icon to add an event.
6. Drag and drop the **[T] Touchable Button** GameObject into the empty field of this event.
7. From this event's drop down list, select ButtonTouchable > ReleaseButton()

8. Now if we play the scene, and click on our button, we should see the button model changes position (revealing a red light) and plays the click sounds as the hand presses it. Just like a proper button!

Make the Button Do something

Now that our button acts like an actual button, the last part is to make it do what real buttons do and that is to make it do something. Let's make it play a car horn sound when we press it. (See the **Getting_Started_6** scene for the completed steps)

1. Create a new GameObject anywhere in the Hierarchy. Rename it To **HORN Player**
2. On the **HORN Player** GameObject, add an *AudioSource* Component
3. Also add the *Ent_AudioClipPlayer* component from Assets \ KINERACTIVE \ Entities in the Project window or under KINERACTIVE > Entities from the *Add Component* button.

4. In the *Audio Clip Player* component, we want to add the *CarHorn* sound file from Assets \ KINERACTIVE \ Sounds
5. Also make sure that in the *AudioSource* component, that *Play On Awake* is unticked.

6. Now go back and find the the **[T] Touchable Button** GameObject and select it.
7. In the *Button (Touchable)* component, find the section called *Button Pressed*
8. Press the little **+** plus icon and add a new event to this section of the component.
9. Drag and drop the **HORN Player** GameObject into the empty field.
10. From the function drop down list, select End_AudioClipPlayer > PlayAudioClip()

11. Play the scene, and press the horn button - you should now hear a honk. However, it will stop honking as soon as the sound clip reaches the end of the file. We want it to keep honking over and over as long as we hold the button down.

12. Go into the **[I] Button Input** GameObject and in the *Input* section, tick the *Repeating Input* checkbox. This will repeatedly trigger the honk over and over.
13. Play the scene again, and hold the button down as long as you can stand the horn, and it will keep on playing over and over *(note: you will probably want to use a sound effect that is designed for a smoother loop in your own project)*

BONUS STEP:

You may have noticed, that when we let go of the horn, it does not stop playing instantly - instead it plays the sound file all of the way to the end even when we release the button. If you do not want this to occur - in the *Button Touchable* component, you can place a new event into the *Button Out* section. Drag in the **HORN Player** GameObject, in the drop down select Ent_AudioClipPlayer > StopAudioClip()

This documentation took me several days to write (and create the scenes). So I know it has taken you a good deal of time to get to the end here as well. So congratulations - the hardest part is over!

Instead of the horn sound event, you could make this do anything, a door opening, monsters appearing, end of the level etc. So once you have your own button set up with a model, and ready positions etc, then you can prefab it and reuse it very easily, and only change what happens in the *Button Pressed* event as you need to.

Summary

We have done quite a lot in this initial tutorial, and because we were so deep in the details, here is a summary so that we can keep in mind the overall picture.

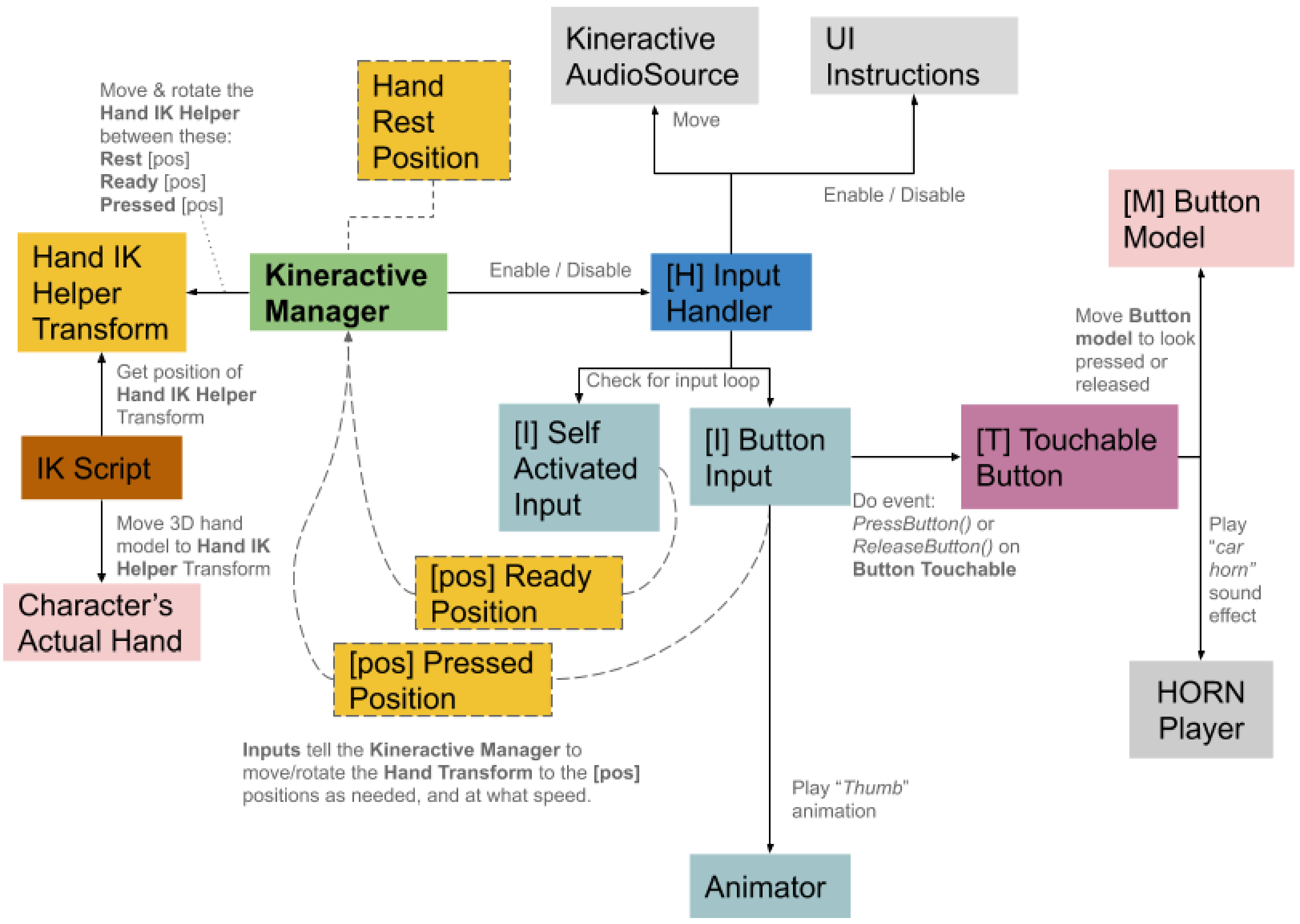
1. We set up the scene for Kineractive with the required prerequisites. Then we added a button model, and made it interactive.

2. As for the button interaction, the most important lesson to take away is that the **Self Activated Input** component moves the hand between the character's knee and the button's *"ready position"*.

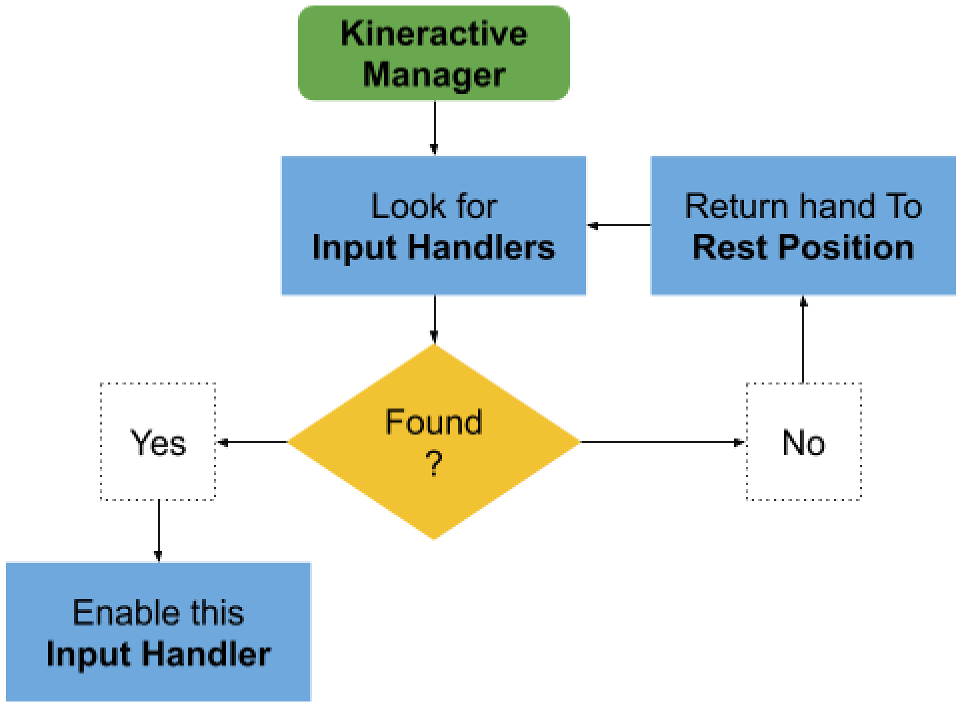
The ***Button Input*** component (when we click) moves the hand between the “*ready position*” and the “*pressed position*”.

3. When our character presses the in game button, the **Button Input** component activates the *Button Pressed* event on our **Button Touchable** component - and when the **Button Touchable** component is pressed in, it activates the play horn sound event on the **Audio Clip Player**.

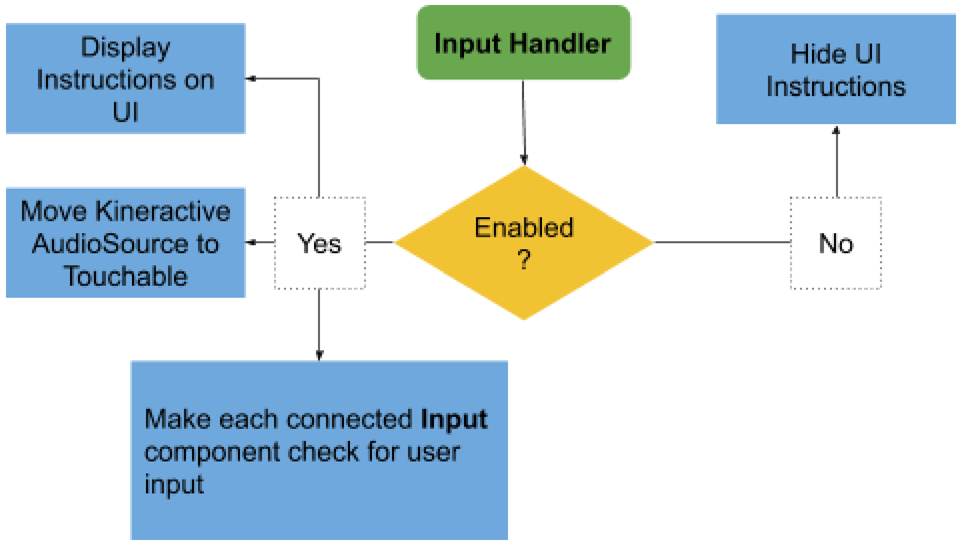
The diagram below shows how the main components are connected and how the events / data flows in our scene:


Kineractive AudioSource
UI Instructions
Move
Enable / Disable
Move & rotate the Hand IK Helper between these: Rest [pos] Ready [pos] Pressed [pos]
Hand Rest Position
[M] Button Model
Hand IK Helper Transform
Kineractive Manager
Enable / Disable
[H] Input Handler
Move Button model to look pressed or released
Get position of Hand IK Helper Transform
Check for input loop
IK Script
[I] Self Activated Input
[I] Button Input
[T] Touchable Button
Do event: PressButton() or ReleaseButton() on Button Touchable
Move 3D hand model to Hand IK Helper Transform
Play "car horn" sound effect
Character's Actual Hand
[pos] Ready Position
[pos] Pressed Position
HORN Player
Inputs tell the Kineractive Manager to move/rotate the Hand Transform to the [pos] positions as needed, and at what speed.
Play "Thumb" animation
Animator

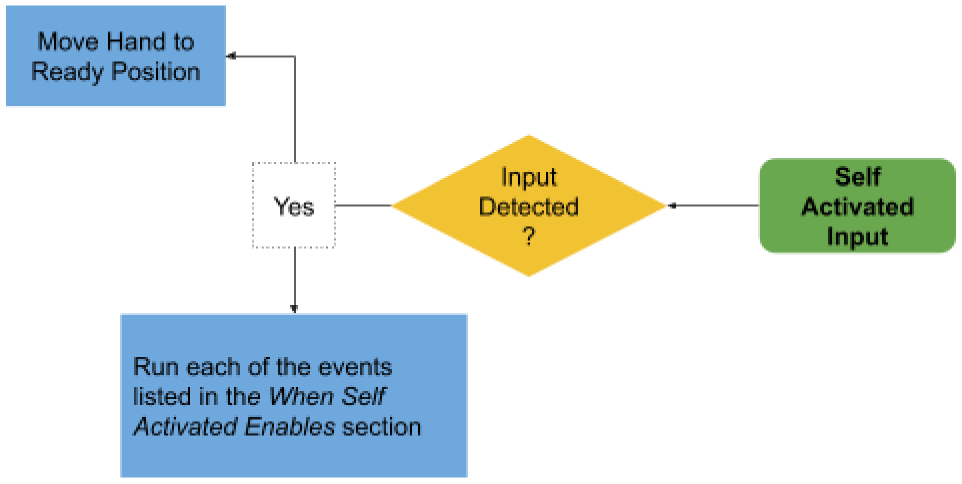
It all starts with the **Kineractive Manager** component. The **Kineractive Manager** shoots out raycasts to look for **Input Handler** components. If none are found, it returns the **Hand IK Helper** Transform to the **Hand Rest Position**. (The **IK** system just follows the **Hand IK Helper** Transform, which is what makes the character's hand actually move).



If an **Input Handler** is detected, then the **Kineractive Manager** enables the **Input Handler**, which then checks every frame for an input from any connected **Input** components, and also displays the **UI instructions**, and moves the **Kineractive AudioSource** to the location of the button (so that when any click sounds are played, they will come from the location of the button - this way we only ever need one **AudioSource** component, no matter how many **Touchables** are in our scene)

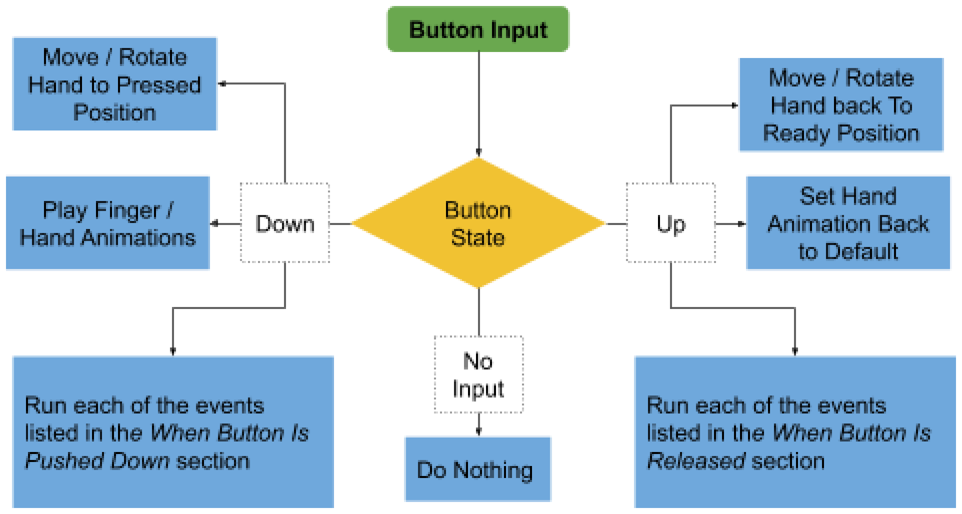

Input Handler
Enabled ?
Yes
No
Display Instructions on UI
Move Kineractive AudioSource to Touchable
Make each connected **Input** component check for user input
Hide UI Instructions

In the same frame, both the **Self Activated Input**, and the **Button Input** check for user input (of course the **Self Activated Input** always gets returns as being pressed/activated since that's the whole point of this component - as soon as the **Input Handler** is enabled, the **Self Activated Input** activates it's events and position moves/rotations)


Move Hand to Ready Position
Yes
Input Detected ?
Self Activated Input
Run each of the events listed in the *When Self Activated Enables* section

The **Button Input** component waits for to receive an input from the allocated Unity Input Manager Button. In this case it is the default **Fire 1** button - which corresponds to the *Left Mouse Click,* or the *Left CTRL* key. When the mouse is clicked or key is pressed. The **Button Input** component runs every event in the When *Button Is Pushed Down* section. It also triggers any animations as selected in the drop down box. And also moves / rotates the hand into the position and rotation of the transform in the *Move\Rotate To* field.

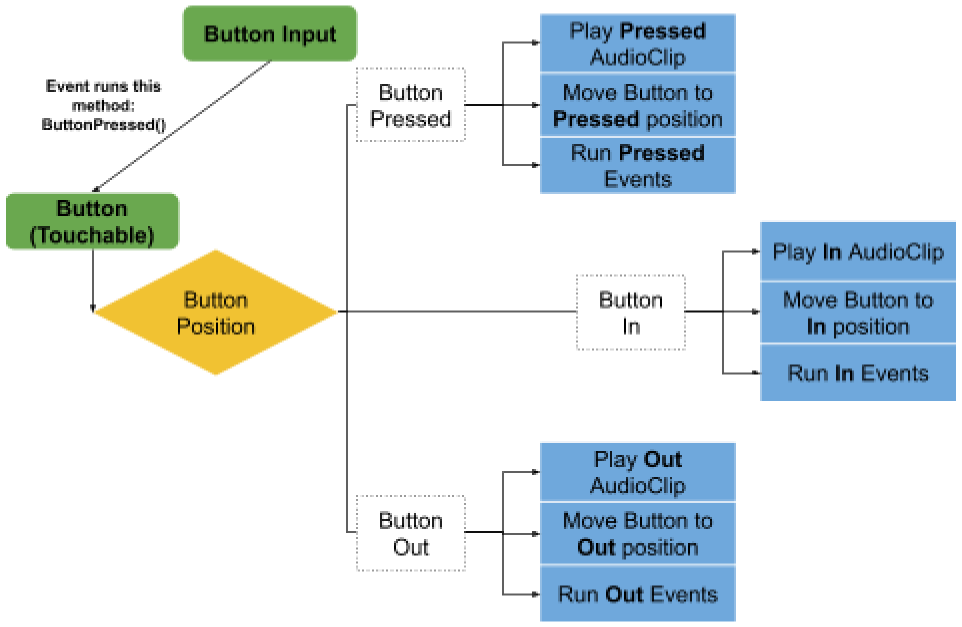
When the button is released, the movement/rotation, animation, and events are run in the *When Button Is Released* section.


Button Input
Button State
Down
Move / Rotate Hand to Pressed Position
Play Finger / Hand Animations
Run each of the events listed in the When Button Is Pushed Down section
No Input
Do Nothing
Up
Move / Rotate Hand back To Ready Position
Set Hand Animation Back to Default
Run each of the events listed in the When Button Is Released section

On the **Button Input** component we have the *When Button Is Pushed Down* section - we added an event to run the *ButtonTouchable.PressButton()* function. When this function runs, the **Button(Touchable)** component moves the 3D button model inwards to the "pressed" position, so that it appears the thumb of our player is pushing it inwards. And most importantly, play's it's own events - which is to run the *PlayAudioClip()* function on the **Horn Player's** *Ent_AudioClipPlayer* component.

Also on the **Button Input** component we have the *When Button is Released* section, and this runs the *ButtonTouchable.ReleaseButton()* function - which in our scene does nothing (by design).

This way, the character's button presses mimic our own real life, press and release actions on the mouse or keyboard. When we click, the character clicks, the in-game button is pressed, and the horn sounds. When we release our real world button/key, the character does the same, and the horn sound stops playing.



There's a lot of information here, and the specifics aren't as important as having an understanding of how the 6 different components types play together. In particular, if you understand that the **Input Handler** makes the **Button Input** check for player's input (to press a button/mouse/keyboard), and then the **Button Input** runs an event on the **Button Touchable**, which then runs an event to do "something/anything" in the game world (e.g. open a door, turn on a light).

The chain of events is: **Input Handler** > **Button Input** > **Button Touchable** > **Play Car Horn SFX**

With that chain of events in mind, we can now create almost any kind of interaction we want by swapping out the components for others of their own type. For example, if we wanted to use a lever instead of a button to play the car horn sound, we'd just have to swap the **Button Touchable** for a **Rotator Touchable**, and swap our button 3D model, for a lever 3D model. Then edit the events so that the **Button Input** now changes the lever position in the **Rotator**

Touchable, and then add an event so that when the **Rotator Touchable** is in the "down" position, it plays the car horn sound.

Our scene changes from this:

Input Handler > Button Input > ***Button*** Touchable & ***Button*** Model > Play Car Horn SFX

to this:

Input Handler > Button Input > ***Rotator*** Touchable & ***Lever*** Model > Play Car Horn SFX

*(**Input Handlers**, **Axis/Button/Self Inputs**, & **Touchables** are best put on the same object in order to represent the button/switch/lever in the game world - as shown in the example scenes. Although they can be placed anywhere in the hierarchy as there are no child/parent connections or relationships, it's only done this way to keep things organised, but if you find a way that works better for your project then you can organise things however you need, even putting all of the components onto the same GameObject if that makes it easier for you).*

Tutorials

Check the ***Clean Shirt Labs*** YouTube channel regularly for up to date tutorials and tips on how to get the most out of KINERACTIVE. Now that the version 1.0 is out, I will be making lots of tutorials to show you everything about KINERACTIVE - starting with the basics, and then into the more creative stuff, so please let me know if there is anything you'd like to see, then you can just copy the steps into your game.

https://www.youtube.com/channel/UCW_agiTbCJ10K6-jAGXwBrA/featured

Making Changes

If you need to make changes to how any of the KINERACTIVE components or scripts work, by all means go ahead. However you will get the best results if you use inheritance to extend any classes, instead of changing them directly. Make a new c# script, and create a new derived class of the class you want to modify. This way the original script stays intact making it easy to reverse the change if required, and in the event of an update to KINERACTIVE, your work won't be overwritten. And feel free to show me what you have done, with your permission I could make it a permanent feature of KINERACTIVE, if that makes things easier for you in the long run.

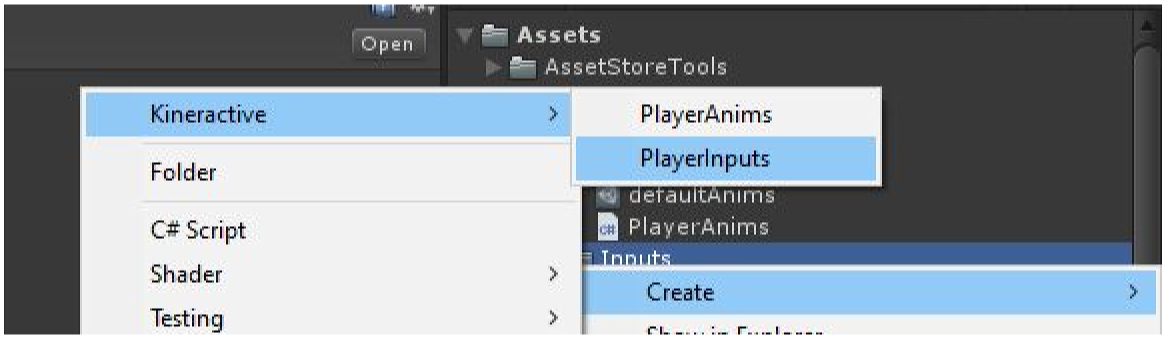
PlayerInputs Scriptable Object

In the **Kineractive Manager** component we have the ability to set which of Unity's Input Manager inputs will appear (as a drop down list) inside the Kineractive **Input** components (Button Input / Axis Input / Analog Input). Which means if you change any of the Input Manager's default input names, such as "**Fire1**" to something like "**Shoot**", then you will need to update existing Player Input files, or create a new one.

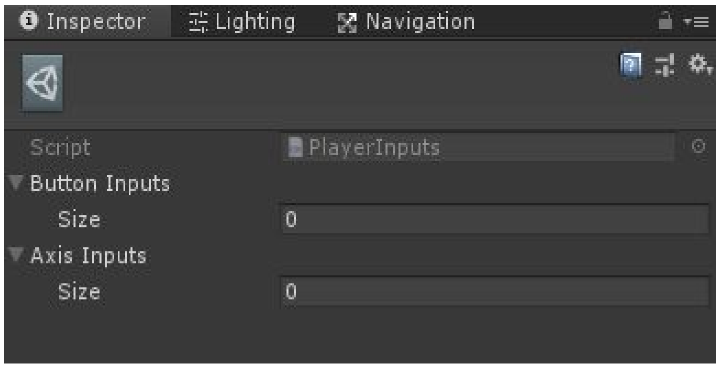
This let's us filter out any inputs we don't want, instead of having to view every one of them from the Input Manager. And it also allows us to change them per scene, by making a new **Player Inputs** Scriptable Object and filling out the details.

Right click anywhere in the Project window and select Create > Kineractive > PlayerInputs

This will make a new **Player Inputs** Scriptable Object file, called *New Player Inputs*.

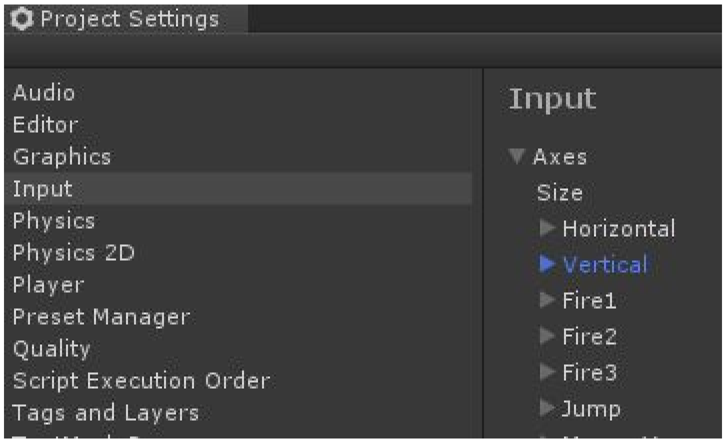


Click on this file, and notice all of the fields are blank.

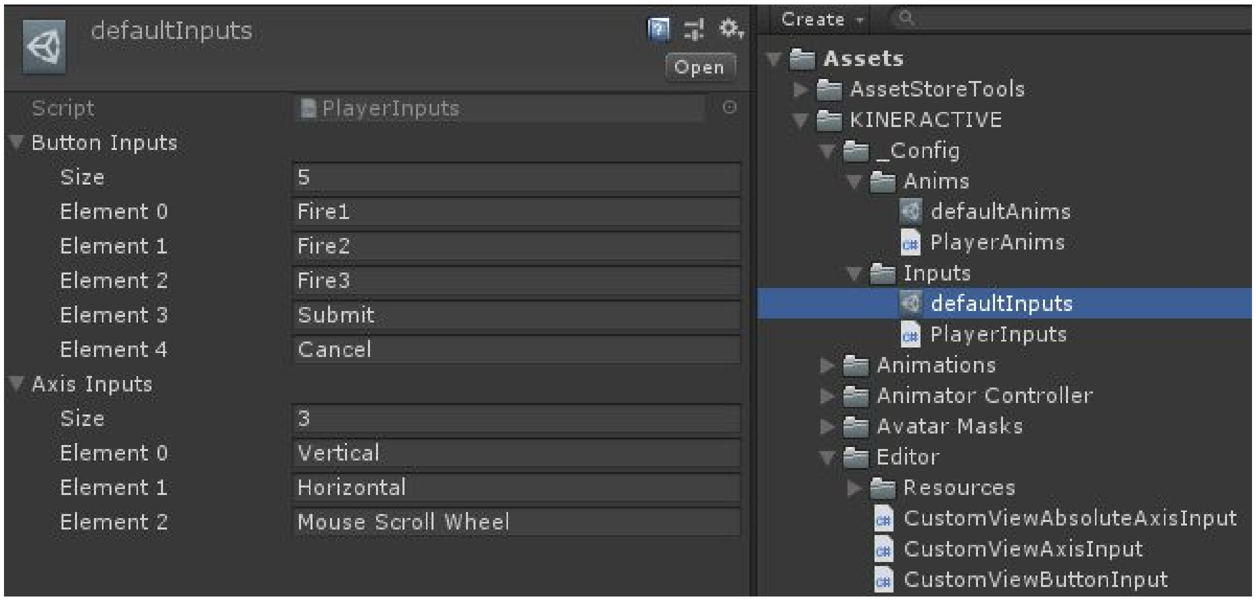


Script PlayerInputs
Button Inputs
Size 0
Axis Inputs
Size 0

Let's change the size to 5, and fill in some names of inputs that we want to use from the Unity's Input Manager.



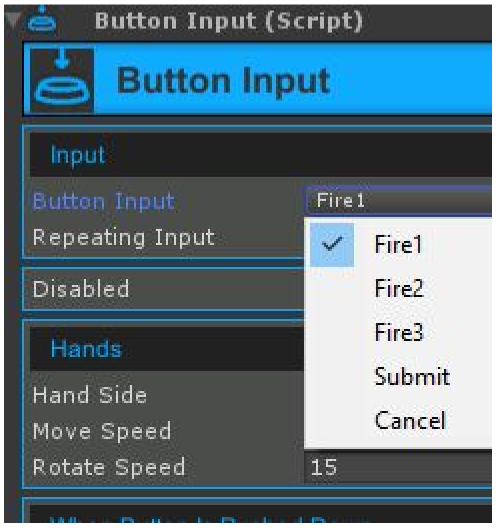
When complete it should look like this default one I made for you.



Script	PlayerInputs
Button Inputs	
Size	5
Element 0	Fire1
Element 1	Fire2
Element 2	Fire3
Element 3	Submit
Element 4	Cancel
Axis Inputs	
Size	3
Element 0	Vertical
Element 1	Horizontal
Element 2	Mouse Scroll Wheel

This way we don't have tons of input options listed that we won't even use. If your project is an action game with only one "use" or "interact" button then you will only need to fill out one element.

We can see that the drop down list matches with what is in the **Player Inputs** Scriptable Object, that we've placed into the **Kineractive Manager's** *Player Inputs* field.


Button Input (Script)

Button Input

Input

Button Input: Fire1

Repeating Input

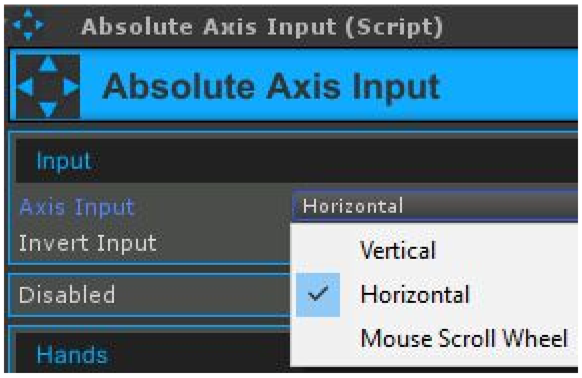
Disabled

Hands

Hand Side

Move Speed

Rotate Speed: 15


Absolute Axis Input (Script)

Absolute Axis Input

Input

Axis Input: Horizontal

Invert Input

Disabled

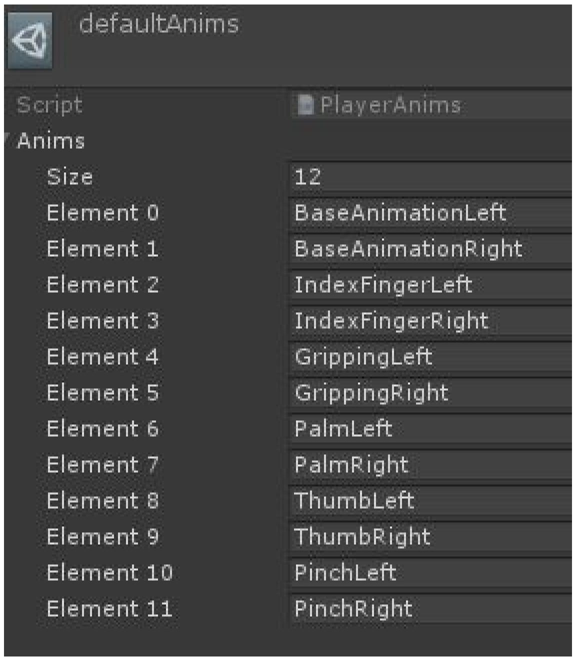
Hands

PlayerAnims Scriptable Objects

Similar to the Player Inputs SO above, the Player Anims Scriptable Object is used to filter in any animation names we will want to appear on our **Input** components. In order for this to work they must match the boolean values of what we have on the Animation Controller of our character. (specifically the **Animation Controller** that we assigned in the **Kineractive Manager** components Animator field).

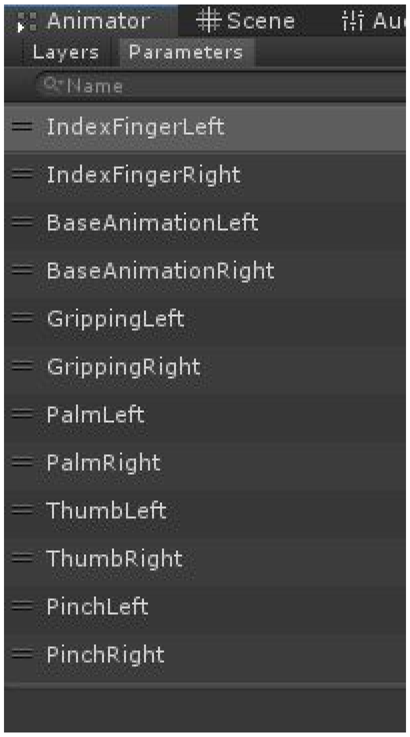
This way we only see the animation parameter names of only the hand animations, and not the walking or sitting animations etc. And it also lets us change the **Animator Controller** and available animations from scene to scene if needed, just by swapping out the PlayerAnims files in the **Kineractive Manager's** *Player Anims* field.

Notice below that the defaultAnims Scriptable Object is matching our boolean Parameters in the Animator Controller. We must make them exactly the same.



defaultAnims

Script	PlayerAnims
Anims	
Size	12
Element 0	BaseAnimationLeft
Element 1	BaseAnimationRight
Element 2	IndexFingerLeft
Element 3	IndexFingerRight
Element 4	GrippingLeft
Element 5	GrippingRight
Element 6	PalmLeft
Element 7	PalmRight
Element 8	ThumbLeft
Element 9	ThumbRight
Element 10	PinchLeft
Element 11	PinchRight

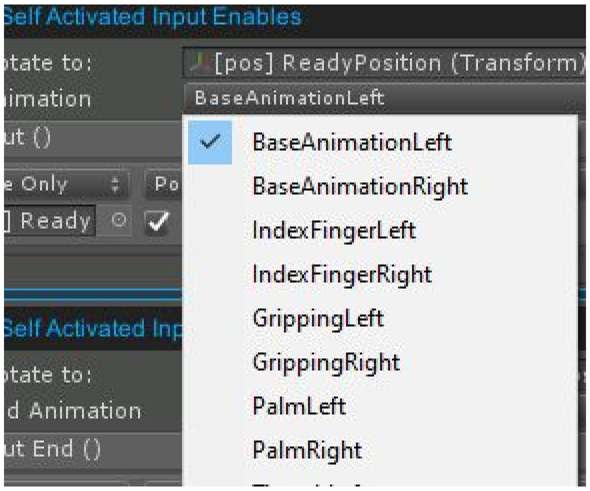


IndexFingerLeft
IndexFingerRight
BaseAnimationLeft
BaseAnimationRight
GrippingLeft
GrippingRight
PalmLeft
PalmRight
ThumbLeft
ThumbRight
PinchLeft
PinchRight

To create a new **PlayerAnims** Scriptable Object file, right click anywhere in the Project window, and select Create > Kineractive > PlayerAnims

Then click on the newly created file, set your size, and fill in the names for the boolean Parameters of your **Animator Controller** so that they match. **They will need to be boolean's** not triggers or floats or ints because that is what the Input components are expecting.

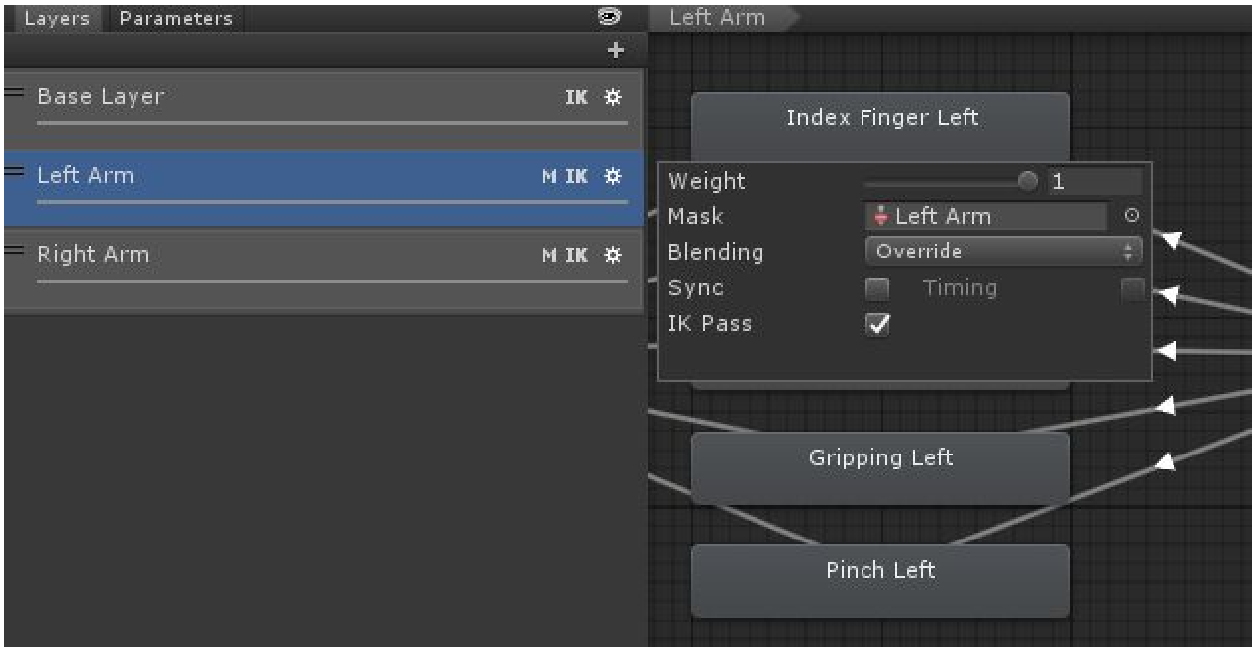
They will then appear in the drop down boxes of any Inputs you put into the scene.



Setting Up Your Own Animator Controller

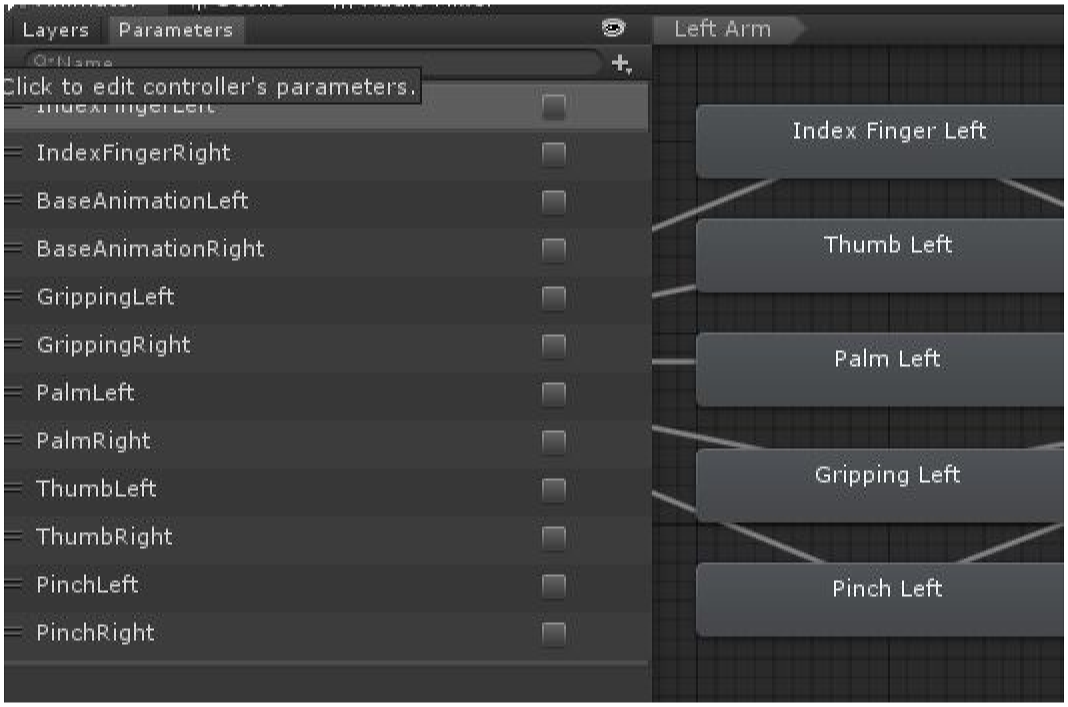
If you set up your own Animator Controller, we need to set something up in a way so that KINERACTIVE is able to understand how to use it. There are unlimited ways to set up an **Animator Controller**, so stick to the 'best practice' recommendations, and incorporate the following settings how it best suits your project. (Or just use the default KINERACTIVE **Animator Controllers** as the base of your own **Animator Controller** and build yours up from this base).

1) You need to have a separate Layer for any hand/arm you intend on animating the fingers on your character's hands.



Weight 1
Mask Left Arm
Blending Override
Sync ☐ Timing ☐
IK Pass ✓

2) Set the Weight to 1 for each arm layer
3) Set the IK Pass to ON/Checked for each arm layer
4) You must use booleans for any finger/hand animation you want Kineractive to enable/disable.


Layers
Parameters
Left Arm
Click to edit controller's parameters.
IndexFingerLeft
IndexFingerRight
BaseAnimationLeft
BaseAnimationRight
GrippingLeft
GrippingRight
PalmLeft
PalmRight
ThumbLeft
ThumbRight
PinchLeft
PinchRight
Index Finger Left
Thumb Left
Palm Left
Gripping Left
Pinch Left

First Person Shooter Mode

In version 1.1 of KINERACTIVE 1st Person support is added. I want to highlight a few of the benefits of using KINERACTIVE in a first person game, and how it can help you add a feeling quality into your game.

In the 1st Person Example scene, the player is able to move around, interact with buttons, and fire a gun. This is pretty standard FPS gameplay. However there are no actual hand animations for any of it. KINERACTIVE handles it all procedurally.

- When the gun shoots the hands move with the recoil of the gun. (the gun has a recoil animation, but it could easily be procedural too)

- As the player approaches a button, the left hand moves off the gun, and goes into a “ready to press the button” position, and then proceeds to press the button (if the player wants to press it). The right hand moves with the rear grip on the gun as the gun points upwards.

- When the player looks away from the button, the hand moves back to hold the front of the gun.

Normally all of this would require a talented animator to spend time in a dedicated animation program to create these animations, and then set them up in Unity to make it all align with other models etc.

We still have to do the setup process in Unity, but we’ve skipped most of the work required to animate the hands, since it all happens procedurally during the gameplay. And further still, since our animations are procedural, if we change the position of something, or the size of a gun, we don’t need to make a new set of animations, or adjust existing ones to match new buttons or weapons - all we move a few transforms around, and it’s done! It can even happen in real time during the game, the button could be shaking all over the place, but because the hand has a target, it will always press the button without missing.

Change Log

V1.11

Added new feature: Pick up, drop, push, throw objects.

Added new script ItemGrabber

Updated Input Handler & Kineractive Manager scripts to allow for setting interaction distance individually for each interactive item, instead of just using the maximum raycast length.

Added new example scene: “Examples Object Pickup”

V1.1

Added 'Out of Box' Support for FPS games (you can now walk around in first person and interact with controls)

Added First Person Shooter example scene (video tutorial coming soon)

Added Counter script

Added Repositioning & Coordinate Sender scripts

Added Enable On Start script

Fixed rotation and position sway so that they use local settings instead of global

V1.02

Removed MIXAMO assets (download available separately)

Added Camera Zoom script with many config options

Added Example scene with camera Zoom script

Added Support for mouselook sensitivity to be scaled based on zoom

Fixed bug with stick handle in Analog Rotation Scene (missing X axis hinge)

Fixed bug with AbsoluteRotation() not triggering 'min/max' & 'out of min/max' events

Fixed bug with AbsoluteRotation() audio

Fixed bug with AbsolutePosition() not triggering 'min/max' & 'out of min/max' events

Fixed bug with AbsolutePosition() audio

V1.01

Added new Component “**Instruction Changer**” - allows for icons to be changed as well as text

Added Input Handler support for Instruction changer

Added Compound Touchables example scene “aka Metroid Prime 3 handle”

Added *Toggle Bypass()* method to all **Kineractive Inputs**

Added Additional sliding sounds

Added Events in Analog Rotator to trigger when rotation has left min or max rotations

Added Events in Analog Mover to trigger when position has left min or max positions

Fixed Rotator Analog now respects min / max without going over/under

Fixed Mover Analog now respects min / max without going over/under1

Fixed bugs in Rotator Analog Example Scene

Fixed bug where **Input Handler** *text* was not changing in the UI until **Input handler** was re-enabled (i.e. by looking away)