player-x

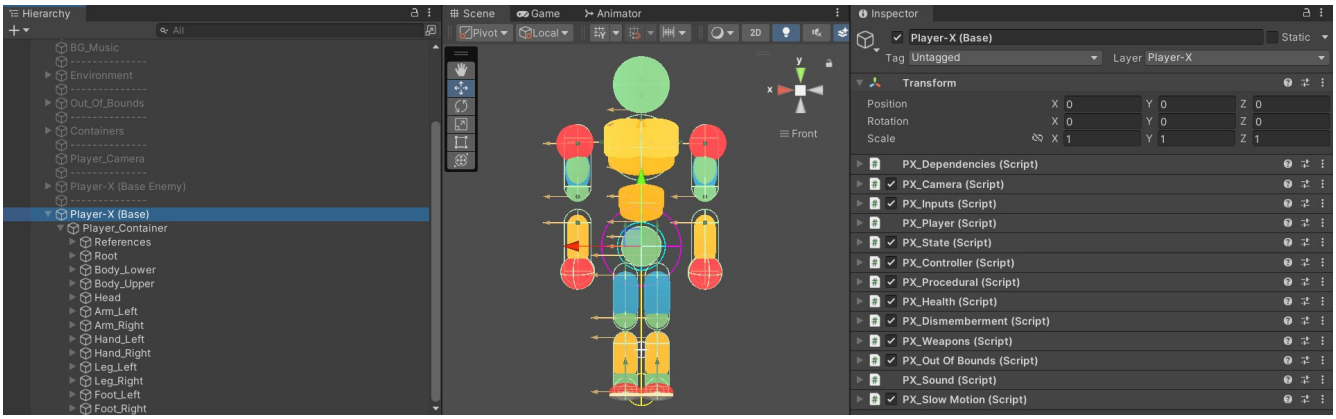
active physics framework

by
the famous mouse

support:
thefamousmouse.developer@gmail.com

Player-X is an active physics base framework for creating physics driven character games similar to popular titles such as Gang Beasts, Human Fall Flat, Totally Accurate Battle Simulator etc.

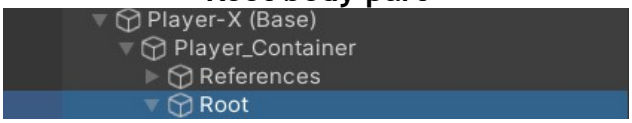
framework



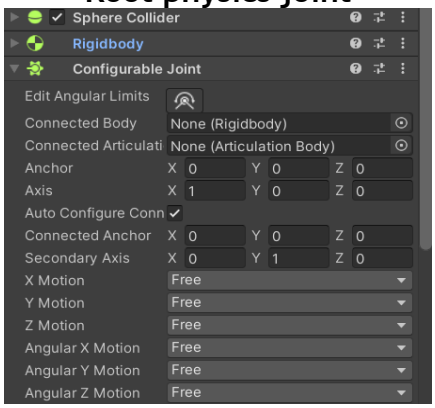
The player structure of our active physics controller consists of 3 main parts.

1. Physics body parts with configurable joints
2. Scripts that control those physics/joints
3. Visual mesh objects

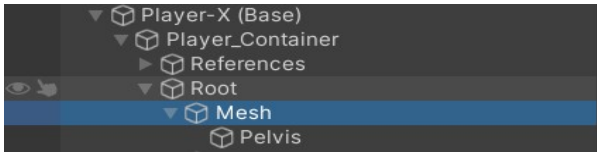
Root body part



Root physics joint



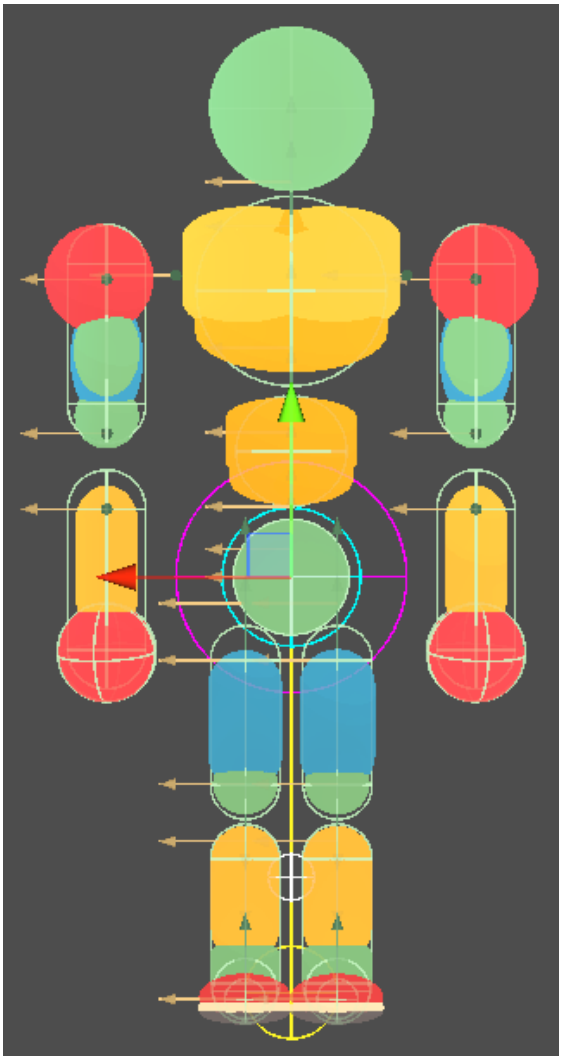
Root Mesh



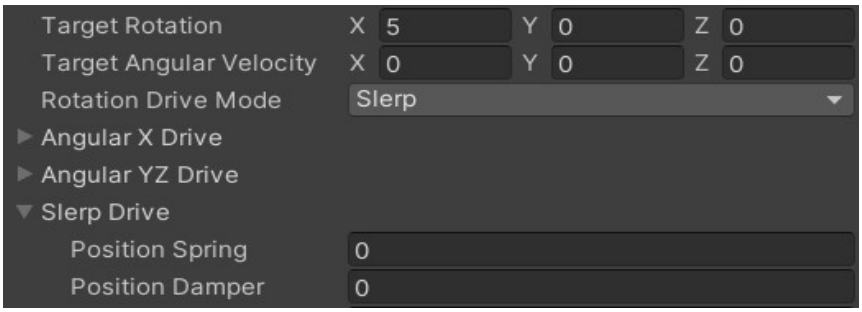
Each body part is connected to its respective other and receives it's active physics qualities from the configurable joint drive.

Body parts

Joints / Mesh



These joints are controlled by the target rotation and it's strength/stiffness by the Slerp drive values.



Character animations are achieved by setting the target rotation values, for example we can set a value while holding down a key and then set another value when the key is released, the Slerp drive strength will correct the rotation to the set value, acting as an auto keyframe between start and end values.

We use this method for all procedural animations, which allows for full physics interaction and interference.

Example of a punch animation:

Key pressed: Pull back arm, curl forearm/hand and twist body pose

```
//... Punch Left
if(dependencies.inputs.keyPunchLeft_Input)
{
    //... Twist body
    dependencies.player.bodyUpperJoint.targetRotation = Quaternion.Slerp(dependencies.player.bodyUpperJoint.targetRotation, Quaternion.Euler(0f, 30f, 0f), 20 * Time.fixedDeltaTime);

    //... Left hand punch pull back pose
    dependencies.player.armLeftJoint.targetRotation = Quaternion.Slerp(dependencies.player.armLeftJoint.targetRotation, Quaternion.Euler(30f, 15f, 0f), 15 * Time.fixedDeltaTime);
    dependencies.player.handLeftJoint.targetRotation = Quaternion.Slerp(dependencies.player.handLeftJoint.targetRotation, Quaternion.Euler(120f, 0f, 0f), 15 * Time.fixedDeltaTime);
}
```

Key released: Set Slerp drive (no pose, our hand gets thrown by force application)

```
if(!dependencies.inputs.keyPunchLeft_Input)
{
    //... Release arms from joint drive
    dependencies.player.armLeftJoint.slerpDrive = dependencies.player.noJointDrive;
    dependencies.player.handLeftJoint.slerpDrive = dependencies.player.noJointDrive;

    //... Left hand punch force forward direction
    dependencies.player.handLeftPhysics.AddForce(dependencies.player.rootPhysics.transform.forward * punchLeftRamp, ForceMode.Impulse);


    //... Restore joint drive
    Invoke(nameof(ResetPunchLeft), 0.2f);
}
```

Set joint drive again (pose will be default value)

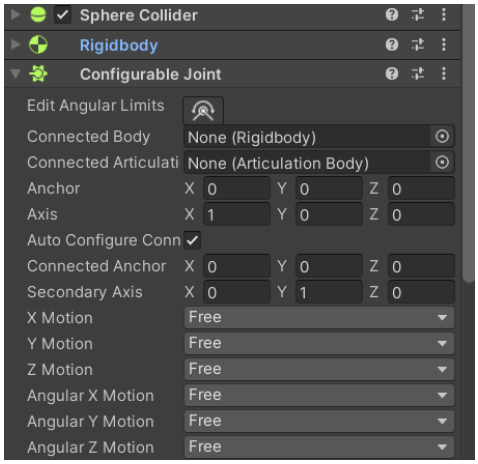
```
//... Reset Punch ...
void ResetPunchLeft()
{
    if(!dependencies.inputs.keyPunchLeft_Input)
    {
        dependencies.player.armLeftJoint.slerpDrive = dependencies.player.armLeftJointDrive;
        dependencies.player.handLeftJoint.slerpDrive = dependencies.player.handLeftJointDrive;
    }
}
```

Note, any collision during these joint animations will then be taken into account which results in realistic/flexible visuals.

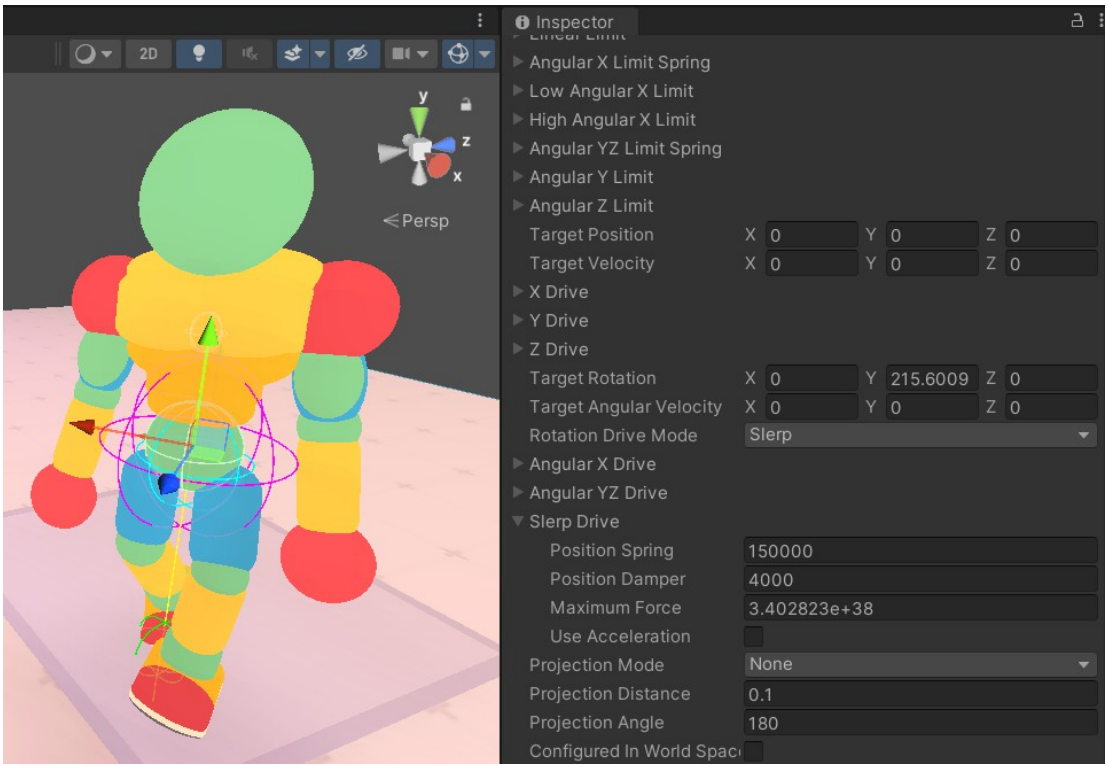
for movement and rotation of the player itself, we use the same principle and rely on both physics Rigidbody and Configurable joint.

We move our player with force or velocity by the Rigidbody (The root body part is used for movement and rotation as all parts are connected to it, themselves or their respective others).

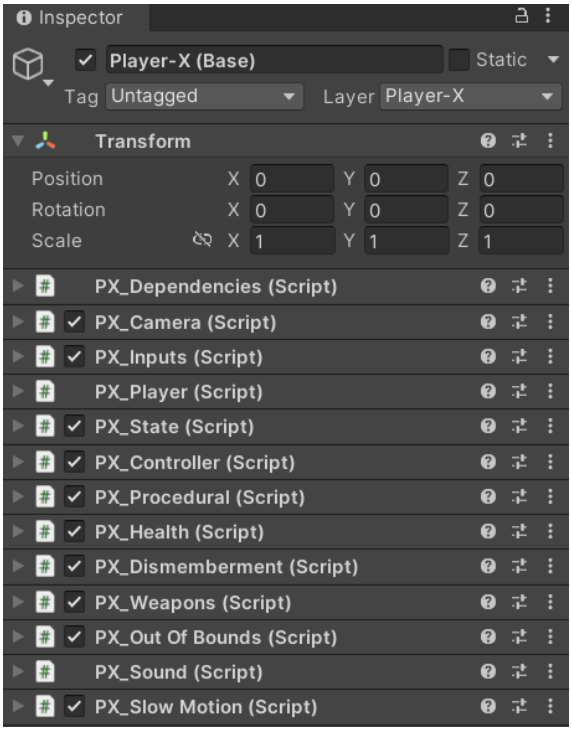
The root joint is set free from all its limits and are connected to the scene.
We apply force/velocity to move and use the joint target rotation to rotate with the help of that Slerp drive strength value.



Root Target rotation and Slerp drive value example.

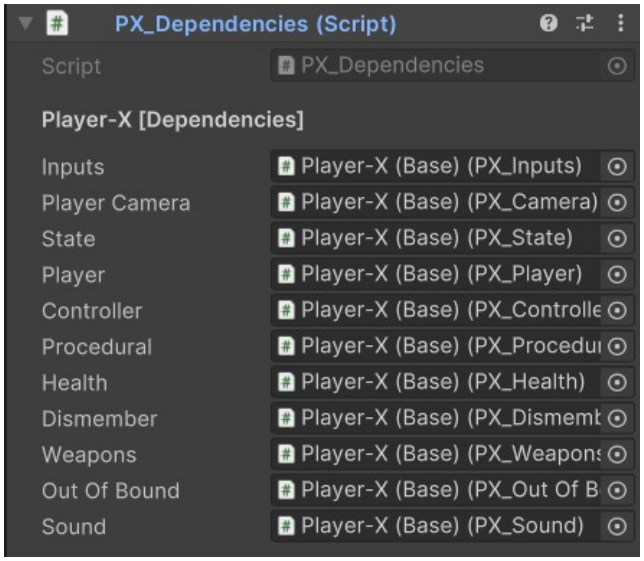


Player-X scripts

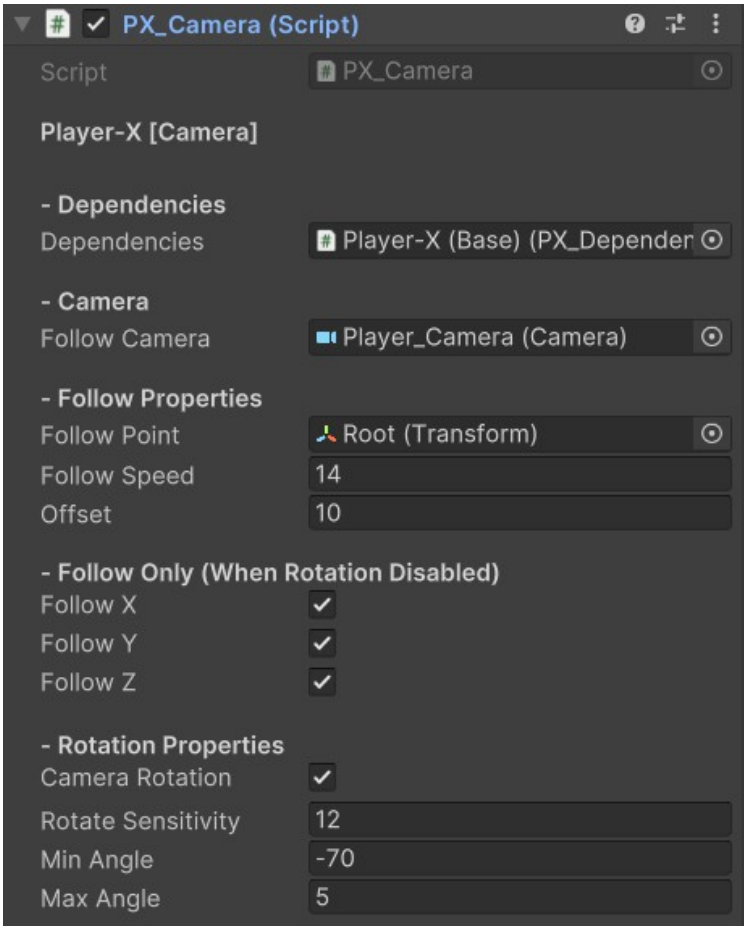


The main character object acts as our player controller and also contains all player body parts, container, layer reference and used for prefabbing.

Scripts are separated but do have dependencies within each other that can simply be removed from the inspector and inside the code if certain features are not required by your game idea. Most dependencies will be boolean variables between individual scripts.

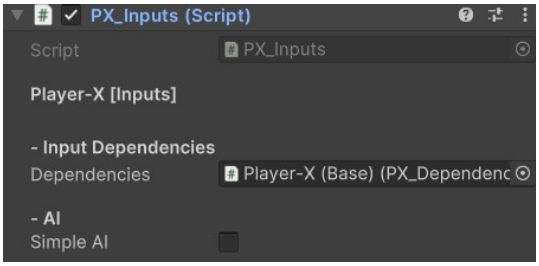


Camera script



For our player we have a camera script that handles the position and rotation of the perspective, we have the option to disable rotation if we intend to have a fixed camera angle and then additional options for limited follow axis.

Inputs Script



The current input script makes use of Unity's new input system but has not yet been setup with the advanced customization, instead directly thru basic code.

However, all inputs have been referenced and knit together neatly for you to change from here so there is no need to go search for every key in other scripts.

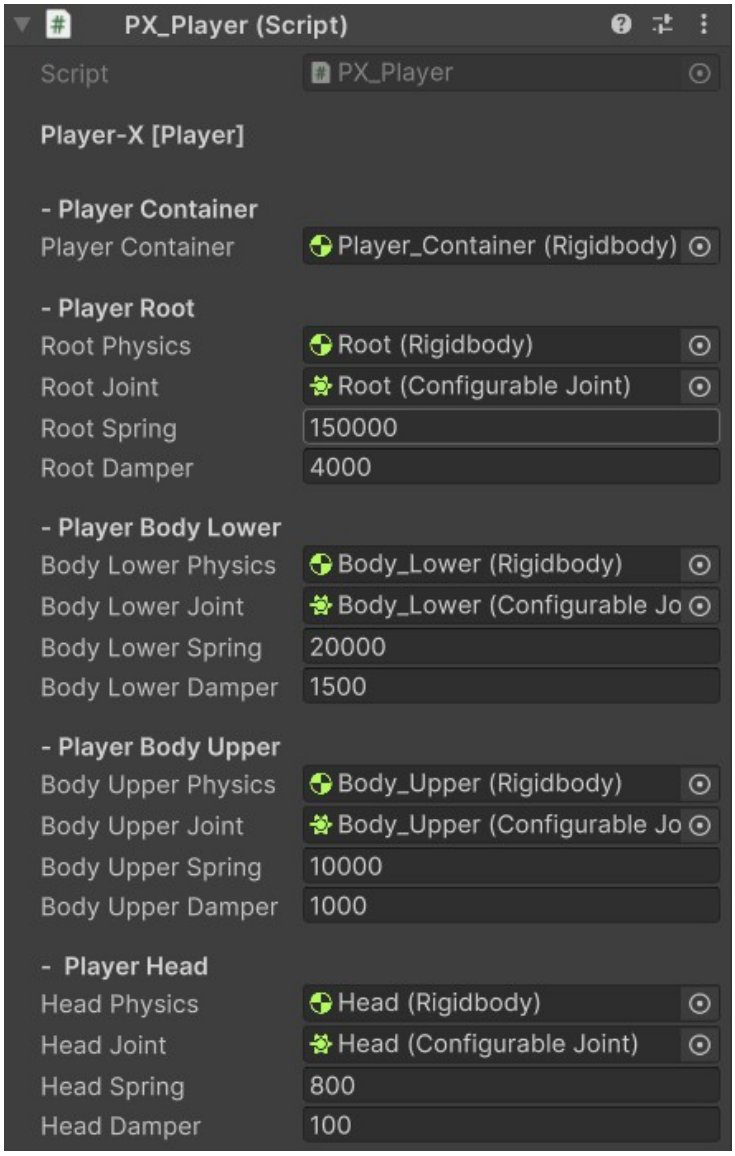
For your convenience, you can simply change the key values without changing the boolean variable or you can add more booleans and use them as reference for actions in other scripts using the dependencies.inputs path.

```
//... Mouse inputs
mouseLeft_input = Mouse.current.leftButton.isPressed;
mouseRight_input = Mouse.current.rightButton.isPressed;

//... Mouse output
mouse_Inputs = Mouse.current.delta.ReadValue();

//... Key inputs
keyLeft_Input = Keyboard.current.aKey.isPressed;
keyRight_Input = Keyboard.current.dKey.isPressed;
keyForward_Input = Keyboard.current.wKey.isPressed;
keyBackward_Input = Keyboard.current.sKey.isPressed;

keyJump_Input = Keyboard.current.spaceKey.wasPressedThisFrame;
keyRun_Input = Keyboard.current.leftShiftKey.isPressed;
keyLook_Input = Keyboard.current.fKey.isPressed;
keyKneel_Input = Keyboard.current.leftCtrlKey.isPressed;

keyPunchLeft_Input = Keyboard.current.qKey.isPressed;
keyPunchRight_Input = Keyboard.current.eKey.isPressed;
keyKickLeft_Input = Keyboard.current.zKey.isPressed;
keyKickRight_Input = Keyboard.current.cKey.isPressed;

keyEquipLeft_Input = Keyboard.current.gKey.wasPressedThisFrame;
keyEquipRight_Input = Keyboard.current.hKey.wasPressedThisFrame;

slowMotion_Input = Keyboard.current.nKey.wasPressedThisFrame;

velocityModeChange_Input = Keyboard.current.mKey.wasPressedThisFrame;

restart_Input = Keyboard.current.rKey.wasPressedThisFrame;

exit_Input = Keyboard.current.escapeKey.wasPressedThisFrame;
```

Example use of “restart_Input” reference in another script

```
//... Restart Scene
if(dependencies.inputs.restart_Input && canRestart)
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
}
```

Player Script

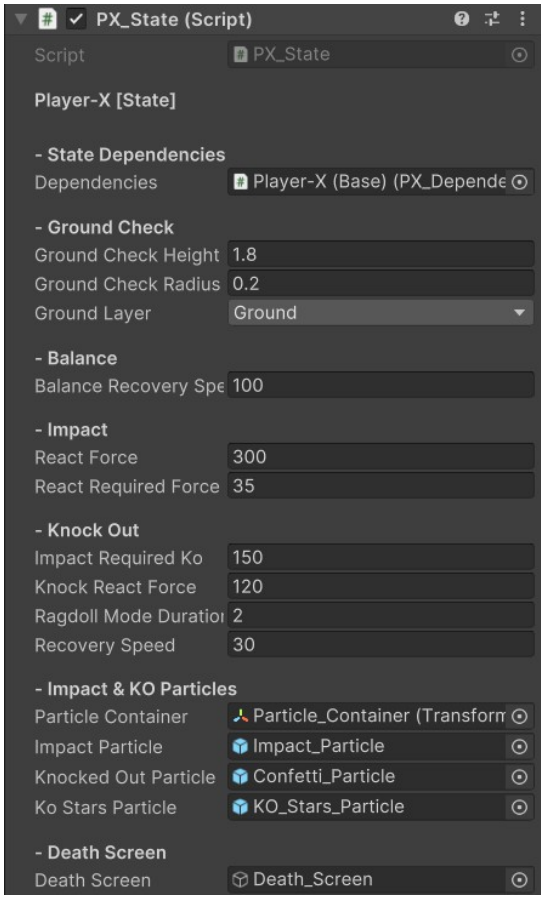


The player script can be considered as the active physics ragdoll character creator, referencing the body parts, joints and it's individual joint drive strengths.

This is what inflates the character and determines it's rigidness, clumsiness and values are also used for switching between active mode (drive/spring value) and ragdoll mode (no drive/spring value).

Additionally, we have and can add more pointers here similar to those used for reaching/grabbing objects, A stronger drive value for the arms when picking things up, stiff enough to perform a pose like punching or as mentioned prior a weaker value to fall into a sloppy ragdoll mess.

State Script

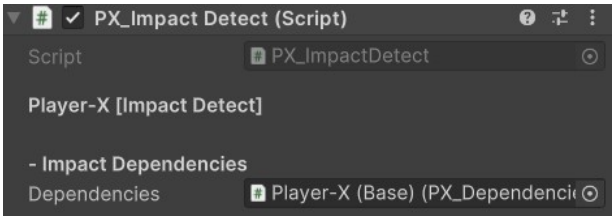


The state of our character affects movement and animations.

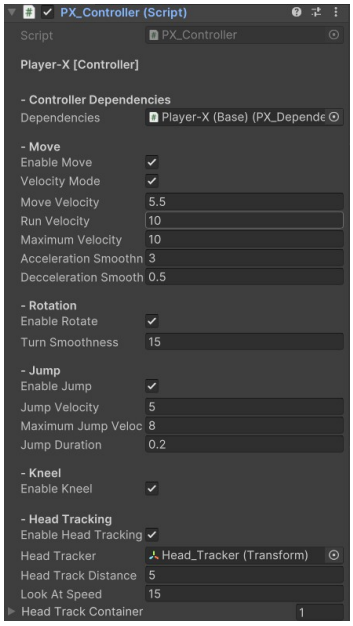
We perform a RayCasting ground check to determine the state of the player, whether it's grounded or in the air, which then in return handle our falling, knockout, active to ragdoll mode and the recovery transition.

Ground objects require the “Ground” layer.

Impacts are detected by a separate script attached to each body part but the reaction is perform here in the state script.



Controller Script



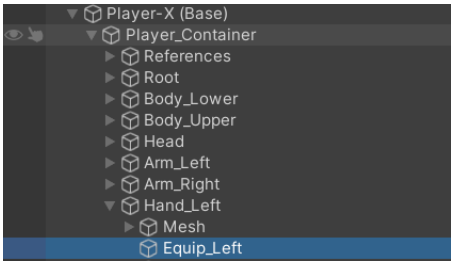
Here most of our player receives it's commands, from movement to fighting.

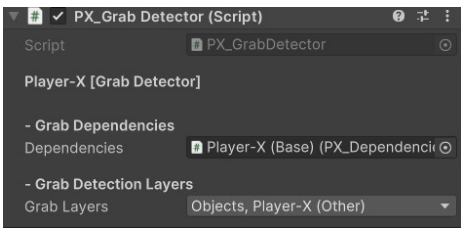
We can move the physics with velocity (accurate but overrides physics) or with force (more clumsy but respect physics).

The head tracking feature can be found here as well and can be used in relationship with the weapons script where the player would look at the weapon only until it has been equipped. Any Object in the head track container will be looked at within range.

Reaching and grabbing objects/other players require the “object” or “Player-X (other)” layer with Rigidbody component to be jointed.

The controller works in relation with the player's hand equip script, grab detector.

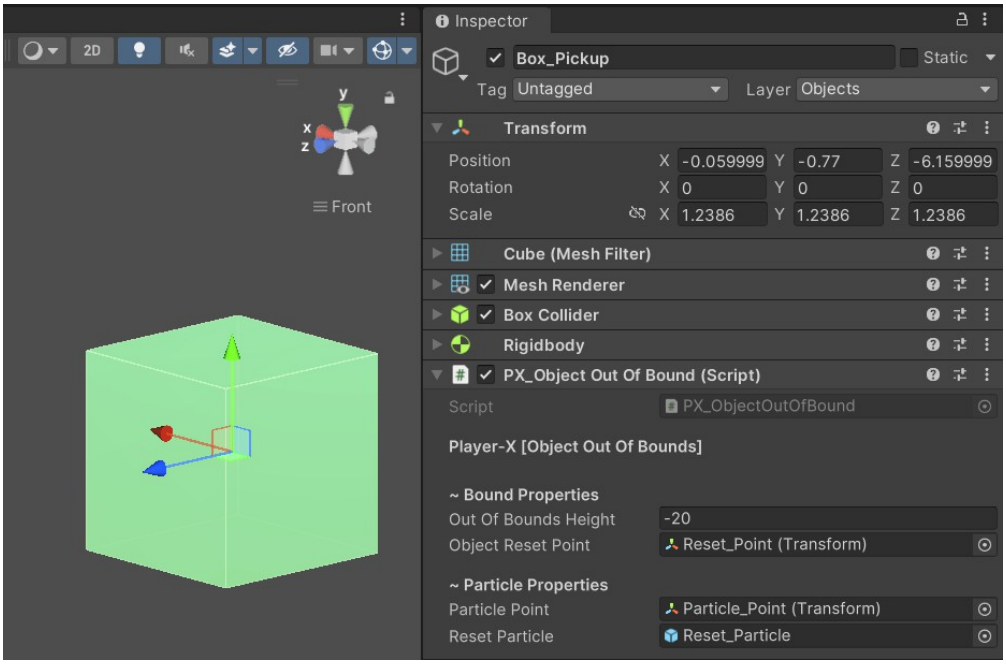




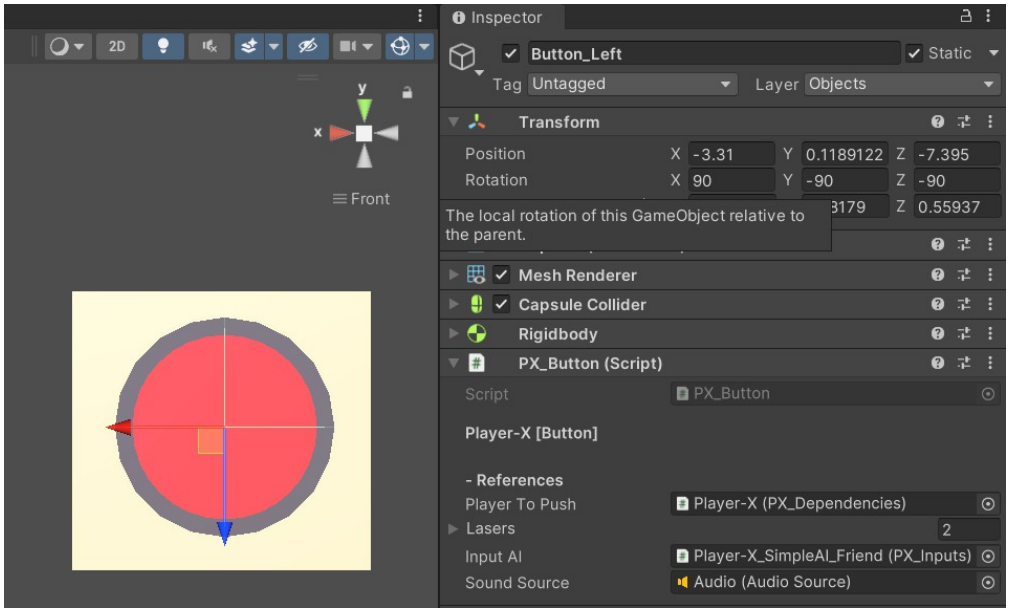
Example of a grab object

The object contains a Rigidbody and are set on the “Objects” layer.

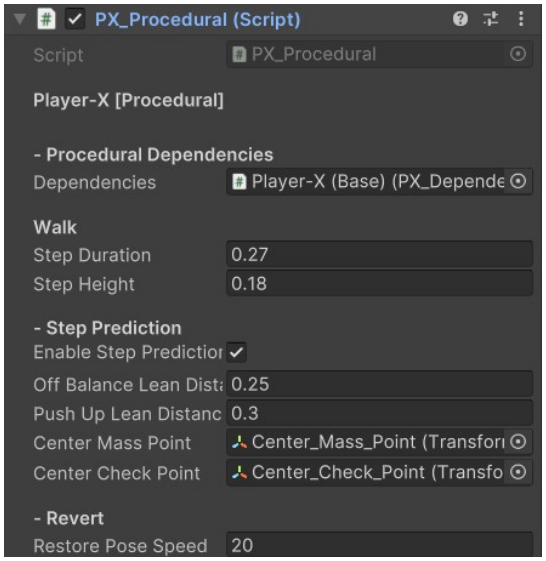
These objects can then be grabbed/jointed to the player's hand and used for placement, puzzles etc.



Another example would be the button object that make's use of both head tracking and grabbing as we include the button object into our head track container the player looks at it when close enough and when reaching to press, the player holds on to avoid pressing/enter and exiting the button's collision multiple times.



Procedural Script

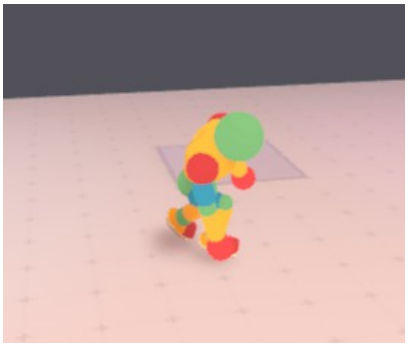


Whenever our player is off balance we use this script to animate it's behavior and visually mimic a step, push and lean pose. Here we handle all joint drive target rotations which does not require input or would not be considered as actions.

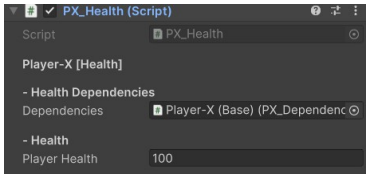
For our movement we take advantage of the stepping by simulating the walk whilst move input is true.

We also revert back to the characters default pose when no input, actions or procedural animations are active.

Off balance correction examples (Step, push, lean)



Health Script

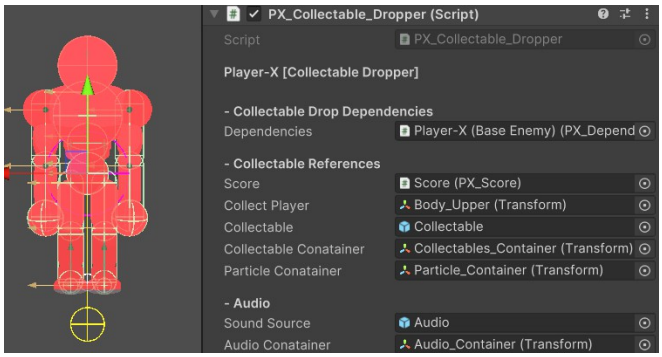


What can I say... health hits zero, player sleeps with the active physics fishes.

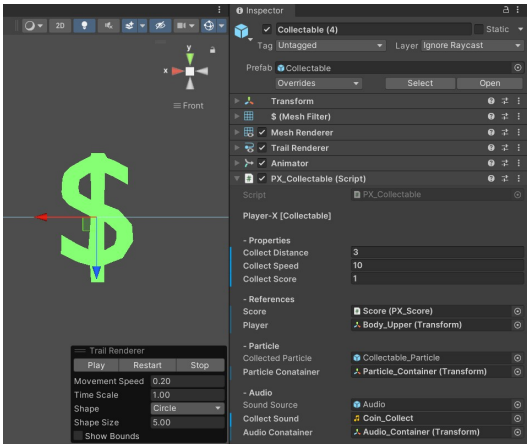
We call for a state change in our state script that indicates whether or not the player is still alive. All input, actions, procedural are halted if the player dies.

We use the state change in accordance with an enemy player for dropping collectibles.

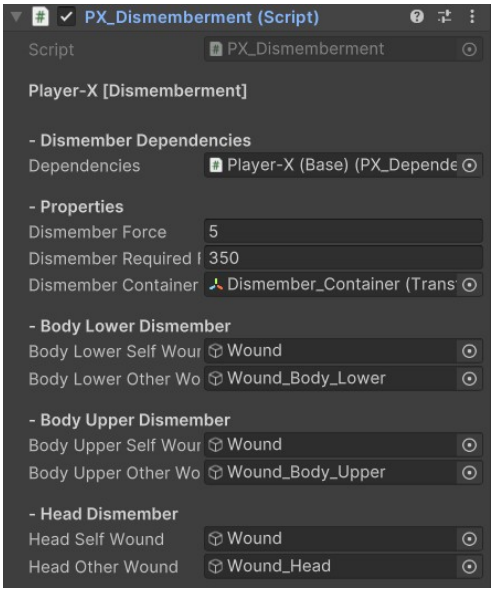
Collectible dropper is an extension for our collectibles that we add to enemies for spawning score points.



Collectibles, score points.



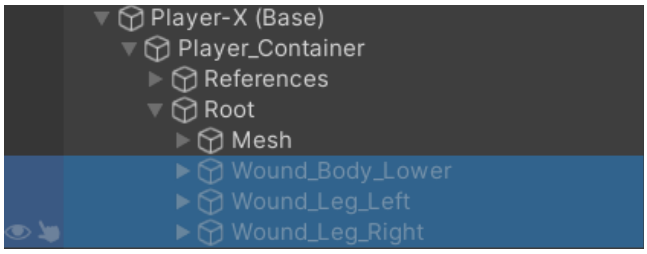
Dismemberment Script

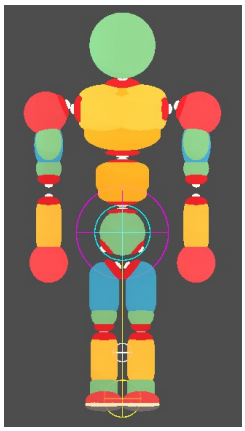


This script contains references to our hidden wound objects that are then enabled whenever a limb is severed on command, by impact force, contact and direct calling.

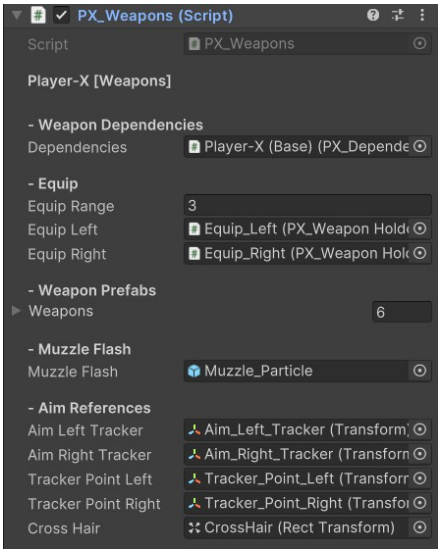
Dismemberment is currently only used for non-Skinned mesh (Individual mesh parts).

We disconnect the limb joint from its connected body whilst making visible of wounds, blood particles.



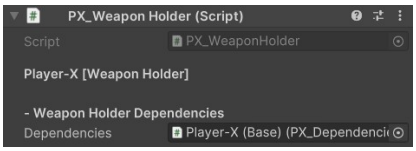


Weapons Script



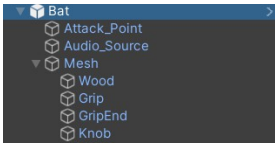
We can equip weapons to the player hands and use our punching action as melee attacks or enable the “isGun” boolean for raycast shooting.

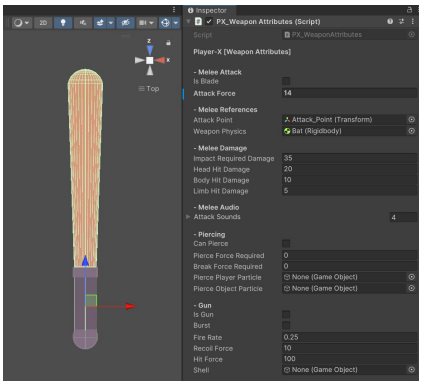
The weapon itself has its own attributes script that contain its unique properties, assigned to our weapon holder when equipped and then used by the weapon script.



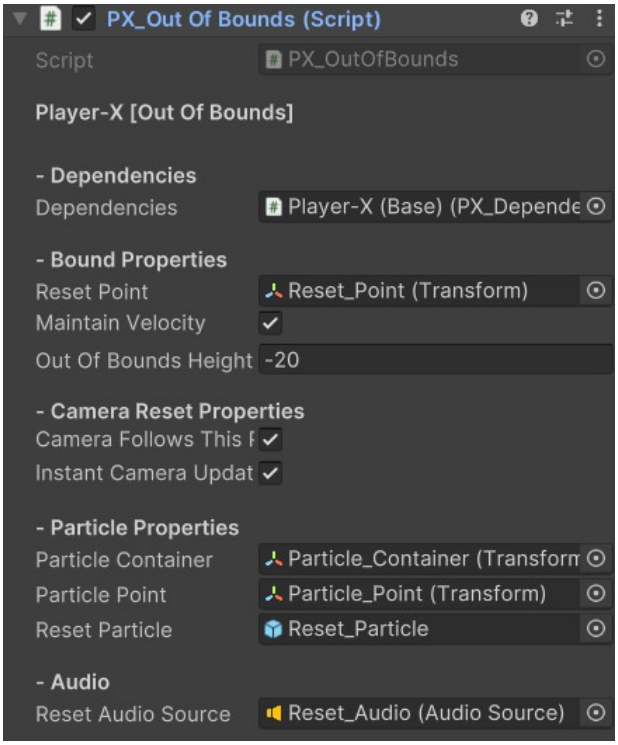
Upon equip, we disable the weapon handle collider, position it to the hand and apply data. (The weapon position will be the root transform, so have it be near the handle)

The weapon has a Rigidbody to be jointed, an attack point reference and if “isGun” a shoot point reference.





Out Of Bounds Script



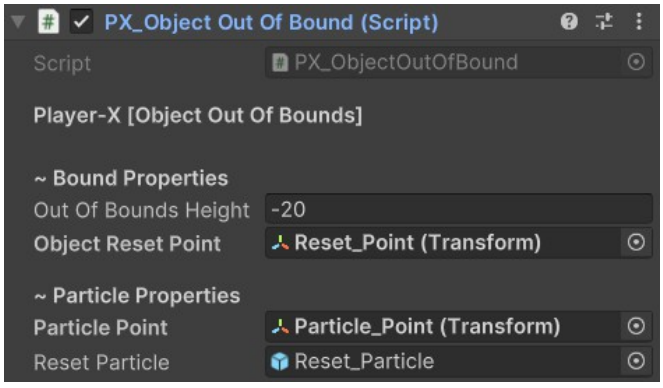
This is just an added little feature used in previous versions which simply re-positions the player back to a starting point when falling off the game scene.

We get all of the player's physics objects, freeze them as to avoid breaking the joints and then move them by the container object.

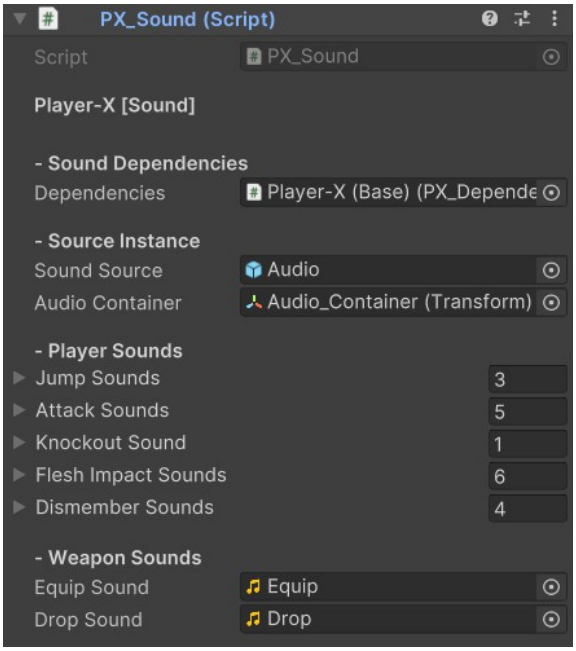
There's an option to record and re-apply the player's current velocity upon out of bound, preserving the fall as appose to dropping it from zero velocity.

Check the instant camera update for continues perspective, else there is the option of letting the camera find its way back to the player over time.

Objects also have their own out of bound script which is used for weapons getting lost amidst the fun.



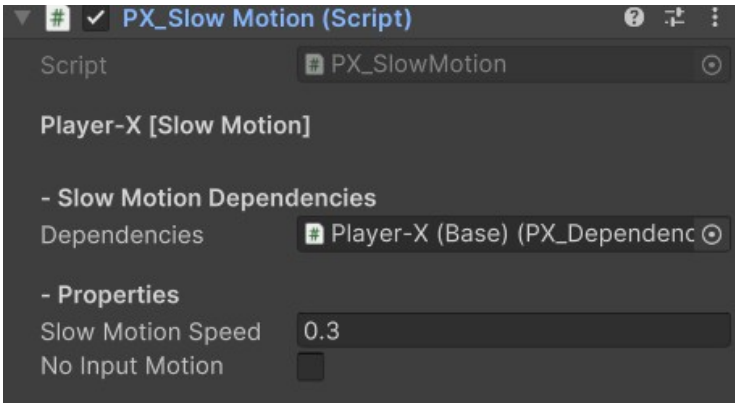
Sound Script



We spawn an audio source prefab which are then used to apply the desired sound to. These prefabs are instantiated within all player scripts.

A random sound is picked from the arrays for variation.

Slow Motion Script

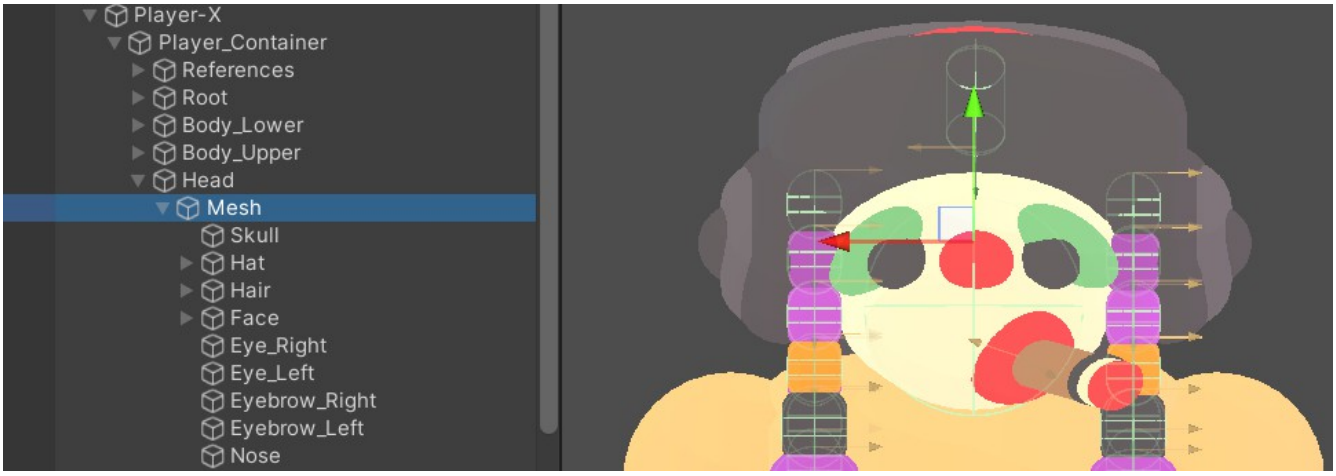


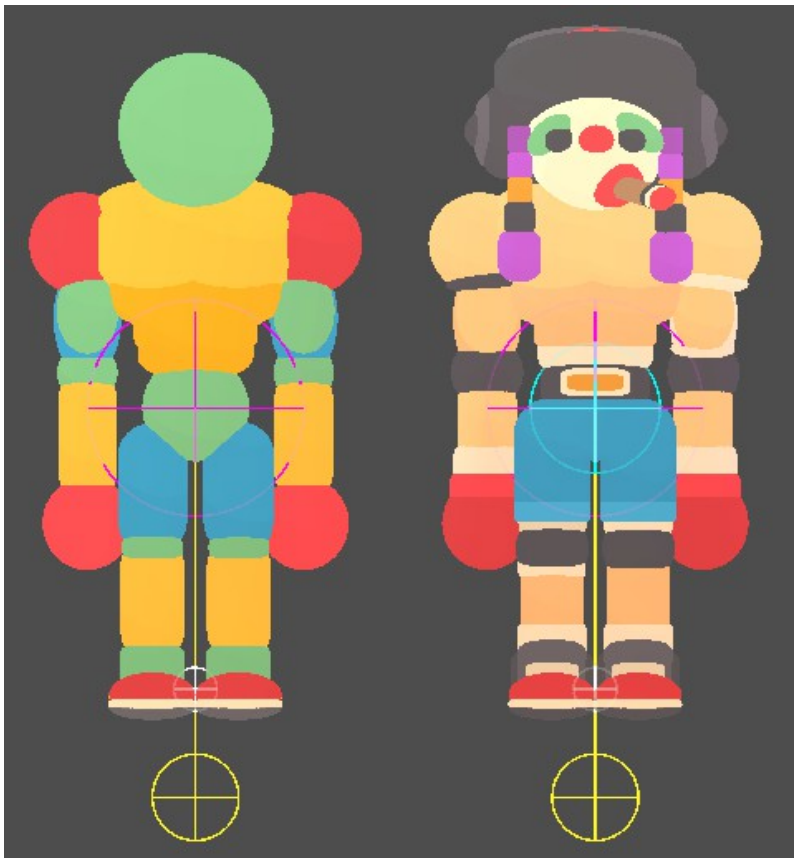
Can't go wrong with some slow motion.
Switch between key command or a bullet time type, where no input results in slow motion.

Changing Character Models

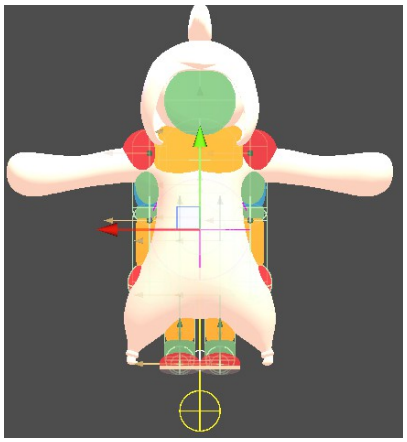
Player-X can be used for both non-skinnedMesh (Multiple individual parts) and SkinnedMesh (Single rigged model).

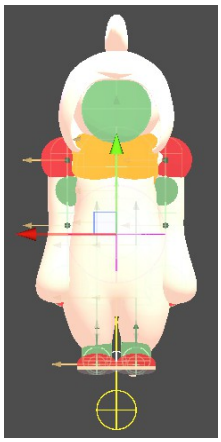
Non-SkinnedMesh models can simply be placed into the mesh container object. What's cool about the current model below is that everything is completely made of Unity's primitive shapes, so there's a quick starting point for changing characters if you suck papaya ass at modeling like me.



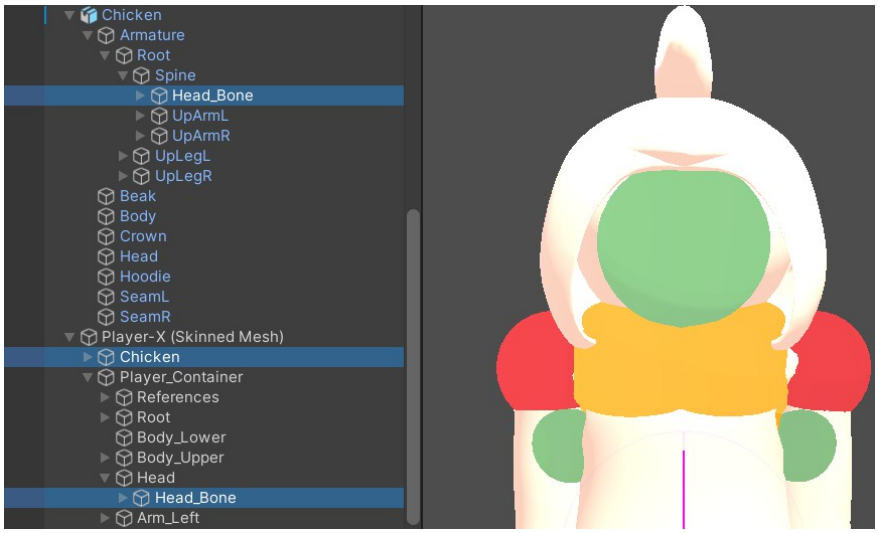


SkinnedMesh rigged models are fairly straight forward too, instead of using the mesh container, we simply overly our rigged model over the Player-X base prefab.

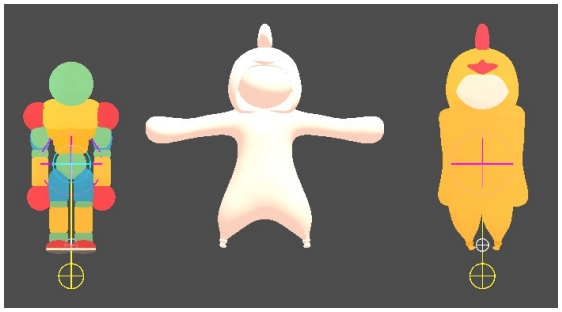




pose the rigged model to match the Player-X base and then drag all the respective bones from the rigged model into the physics body parts. Finally we put the mesh into the Player-X prefab and you should be able to clumsy around with your new skinnedMesh.

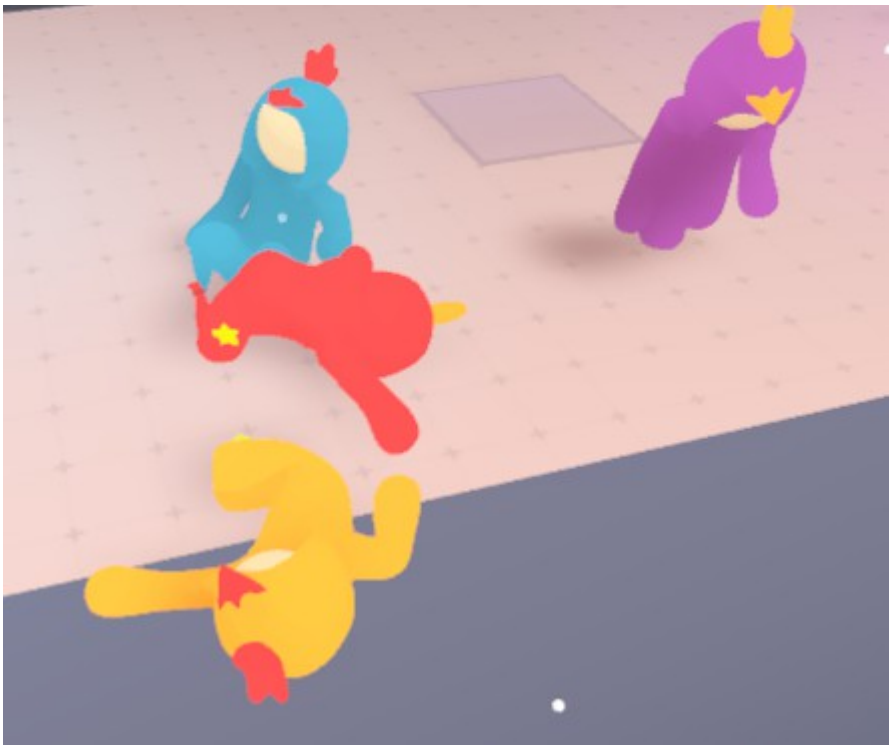


Ideally you want to play around with the collider sizes to better fit your model and tweak additional properties in the inspector.



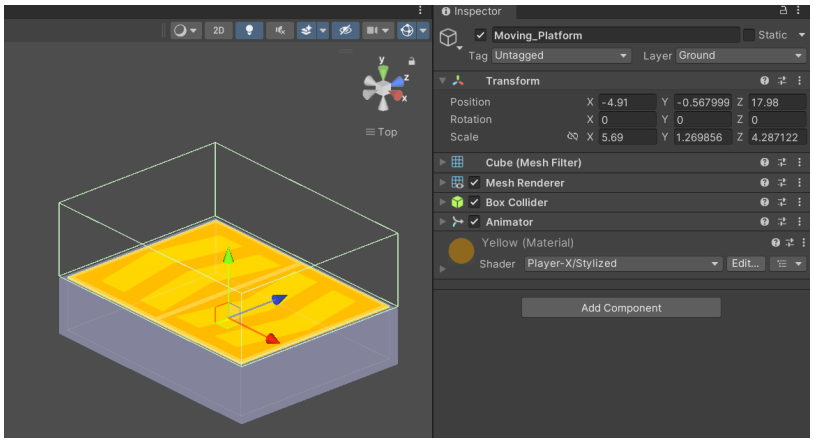
bob's your uncle and you've got active physics

oh... and you must be wondering why it looks so cool?
There's a simple shader included but the magic behind the soft look is a combination of both the shader and low resolution shadows which gives it the same appearance as an orthographic camera...but in perspective mode, wink wink.



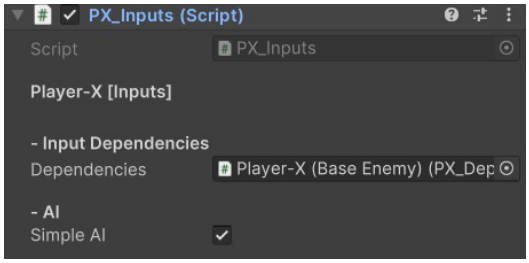
Additional info

Moving platforms are hitch hiked by parenting the player to the platform when entering the zone trigger and vise versa.



Ai is not yet implemented for enemy player's but a simple example was added to help those interested in implementing it.

In the input script we can replace the keys with other methods of controlling the system.



Example of picking a random number that will dictate the Ai action, punch left or punch right. If 0 we set keyPunchLeft_Input to true, the same boolean used for when a key is pressed. This performs the exact same action as player input without re-writing a new system.

```
var actionRondom = Random.Range(0, 2);
actionTime = Random.Range(0.5f, 1.5f);

if(actionRondom == 0)
{
    keyPunchLeft_Input = true;
    Invoke(nameof(PunchLeft), actionTime);
}

else if(actionRondom == 1)
{
    keyPunchRight_Input = true;
    Invoke(nameof(PunchRight), actionTime);
}
```

A more in-depth documentation and updates/features will follow, the scripts do have commented notes which should be helpful in summary of what the snippets are for as well and I hope this helps you get started but if you have any questions, my support email: thefamousmouse.developer@gmail.com

Thank you for your interest and support.

Make fun games.