APR Player

Active Physics Ragdoll Player

https://www.youtube.com/channel/UCJ3AMSuh9axNllOnkznABtw

Documentation

- ## Contents

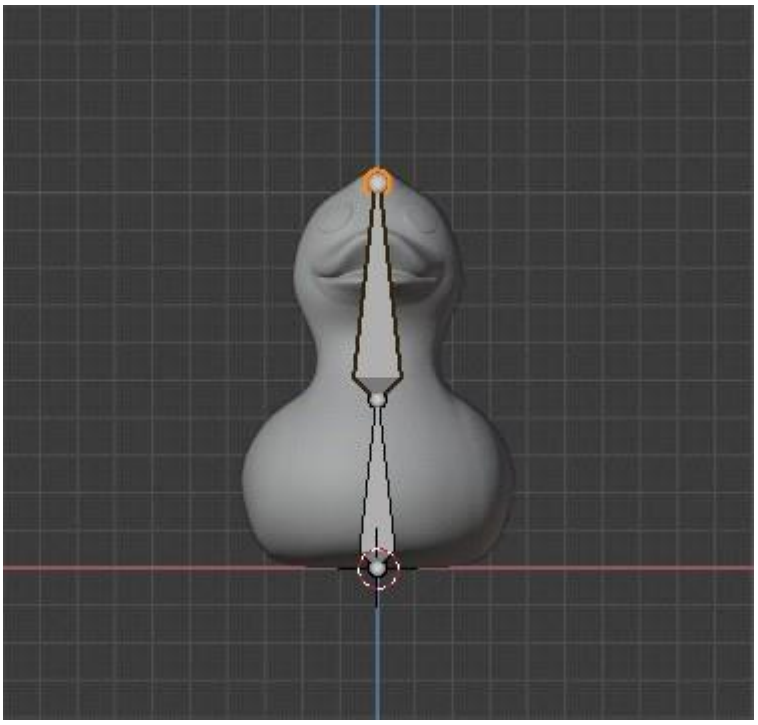
The Concept

Understanding the concept and methods behind active physics ragdolls will open the doors to many interesting game mechanics without writing an insane tweening algorithm or fancy animation composition to impersonate a flock of attacking rubber ducks...as not only is it important to do things as simple as possible for the sake of development time but also save yourself from overly complicated and exhausting code.

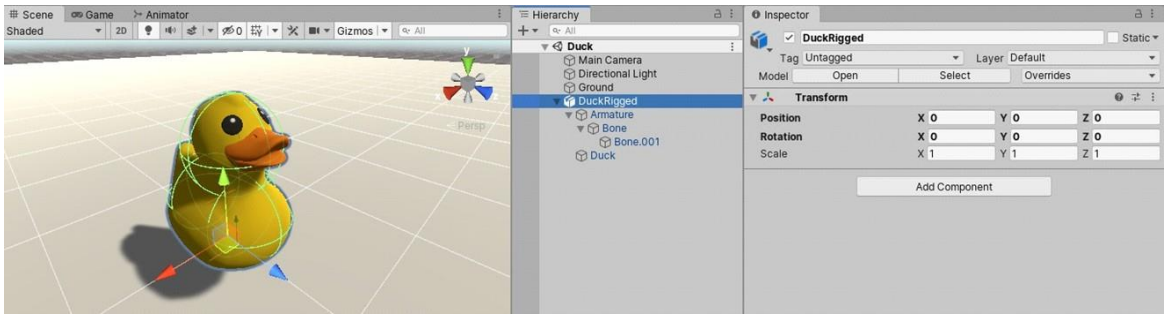
With Unity's build in physics components we can create such behaviors by using them as a foundation instead of an addon.

Here's the concept method by using the attacking rubber ducks as an example.

Using Blender I'm adding 2 bones, the bottom one being the root and the top one for the use of creating a rubber effect.

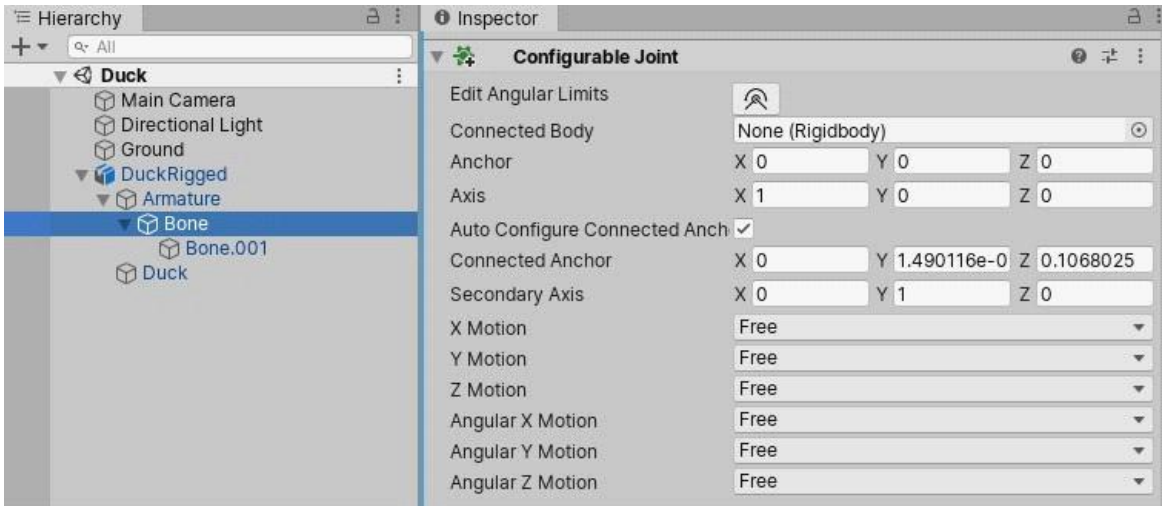


Inside Unity I've added colliders to both bones.

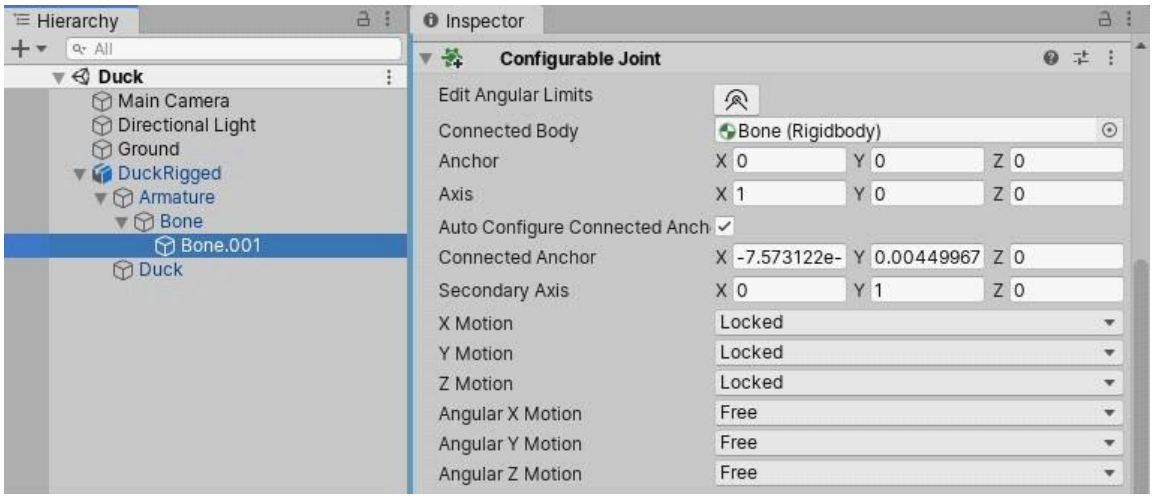


Next I'm adding a rigidbody and a configurable joint to these bones.

The first bone (Root bone) is free and has no restraints or limits.

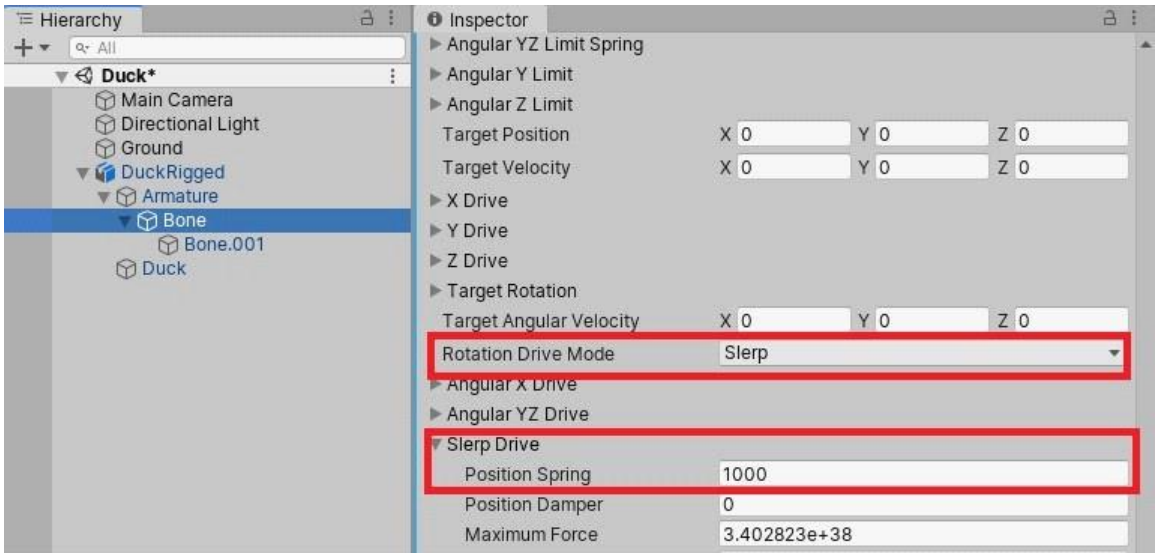


The second bone has locked positions and are connected to the first root bone.

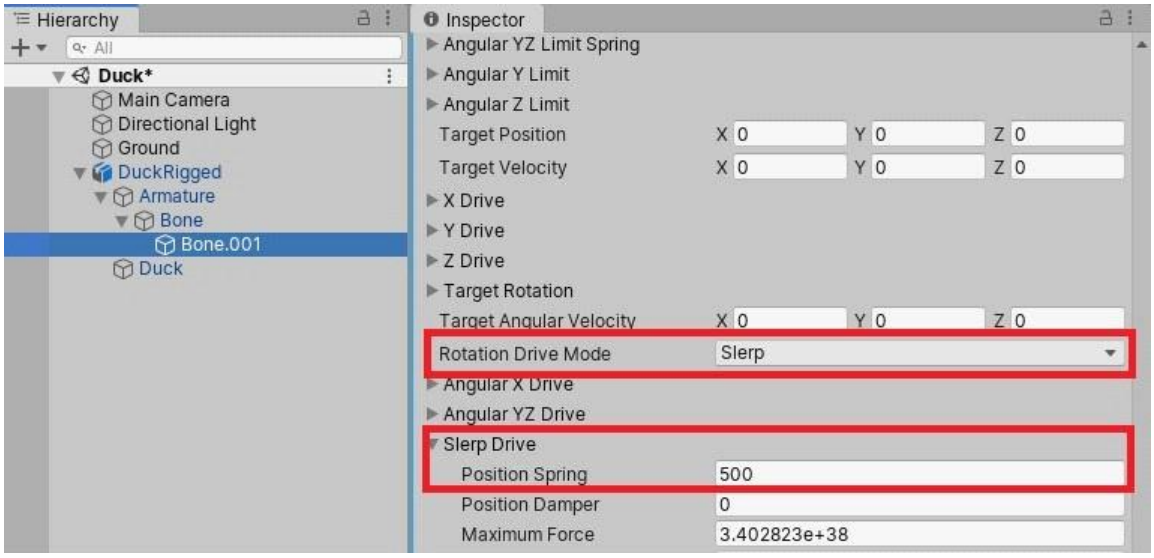


Lastly I'm going to add angular drive to both of these configurable joints, you can either use XYZ or slerp, for this example i'll be using slerp drive which is just one value for all angles.

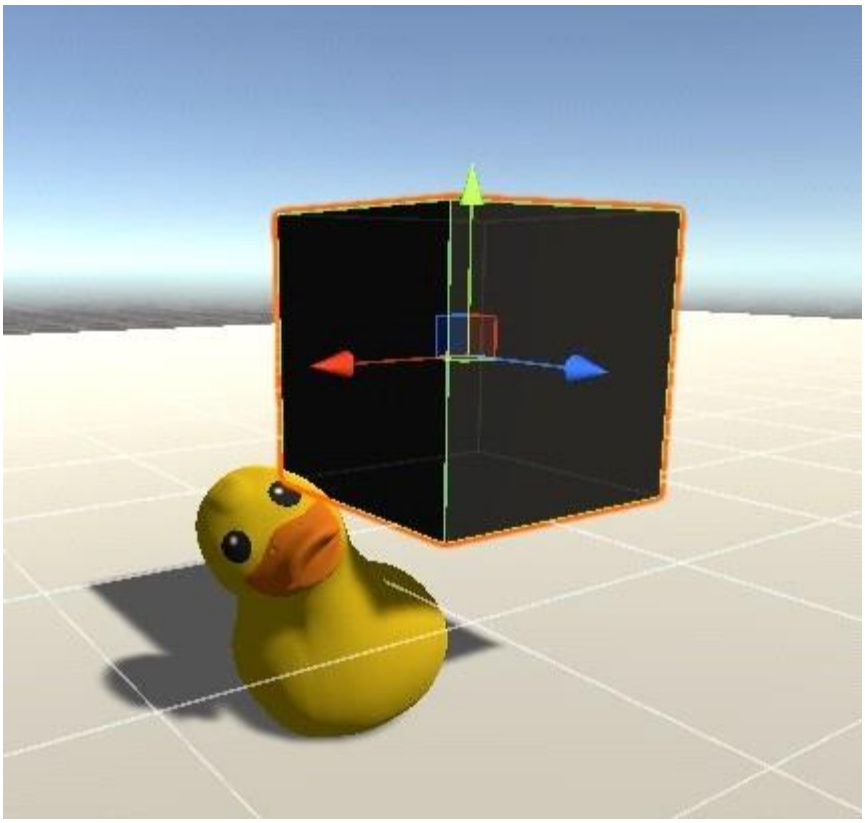
The root bone configurable joint drive settings



The second bone configurable joint settings



I'm making the root bone more stiffer than the top bone, to keep it upright and to create the rubber effect with the loose top.



See animated GIF of this effect here:

https://drive.google.com/file/d/1oti9xT6zSplHaD3Mc5-LnlAzAclTwD_1/view?usp=sharing

Now that I've setup the fundamentals of an active physics character, I can now start adding behavior, with a few lines of basic code we can achieve interesting results.

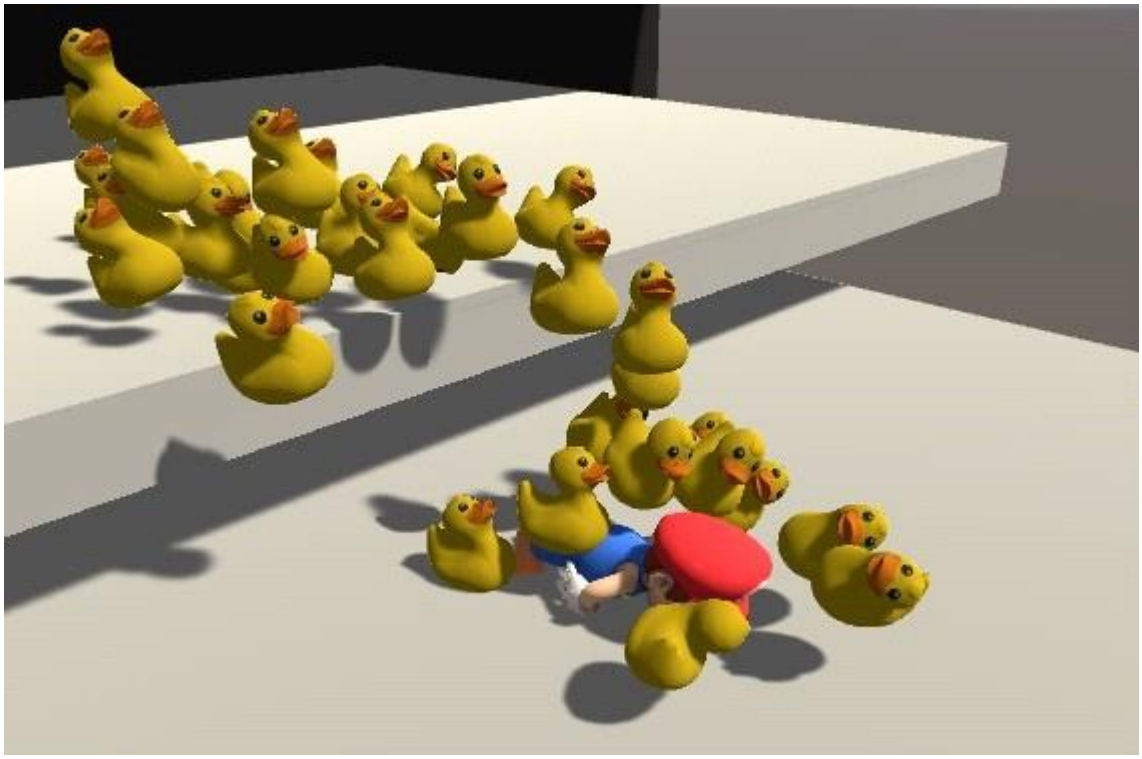
I am adding rotation, bounce and move towards a target

```
public class Duck : MonoBehaviour
{
    public Transform followTarget;

    public float followSpeed = 500f, turnSpeed = 4f, bounceHeight = 20f, bounceRate = 1f;

    private Rigidbody rb;
    private ConfigurableJoint cj;

    private bool bounced;
    private Vector3 moveDirection;


    void Start()
    {
        //Store this duck's rigidbody and configurable joint for future use
        rb = this.GetComponent<Rigidbody>();
        cj = this.GetComponent<ConfigurableJoint>();
    }


    void Update()
    {
        //Direction
         moveDirection = followTarget.position - transform.position;

        //Rotation
        var lookPos = moveDirection;
        lookPos.y = 0;
        var rotation = Quaternion.LookRotation(lookPos);
        cj.targetRotation = Quaternion.Slerp(cj.targetRotation, Quaternion.Inverse(rotation), Time.deltaTime * turnSpeed);


        //Bounce and move
        if(!bounced)
        {
            //Bouncing
            bounced = true;
            rb.AddForce(this.transform.up * bounceHeight, ForceMode.Impulse);

            //Movement
            rb.AddForce((moveDirection).normalized * followSpeed);

            StartCoroutine(DelayCoroutine());
            IEnumerator DelayCoroutine()
            {
                yield return new WaitForSeconds(bounceRate);
                bounced = false;
            }
        }
    }
}
```

- *I've attached this script to the first root bone*

That's it, the concept behind active physics characters. The creative possibilities are up to you.

Here's the result of the above steps...yes...A flock of attacking active physics ducks!



See animated GIF of this effect here:

https://drive.google.com/file/d/1VSD0WvI9XvI6V9N5OKgEw5Eh7sBLWRsu/view?usp=sharing

How It Works

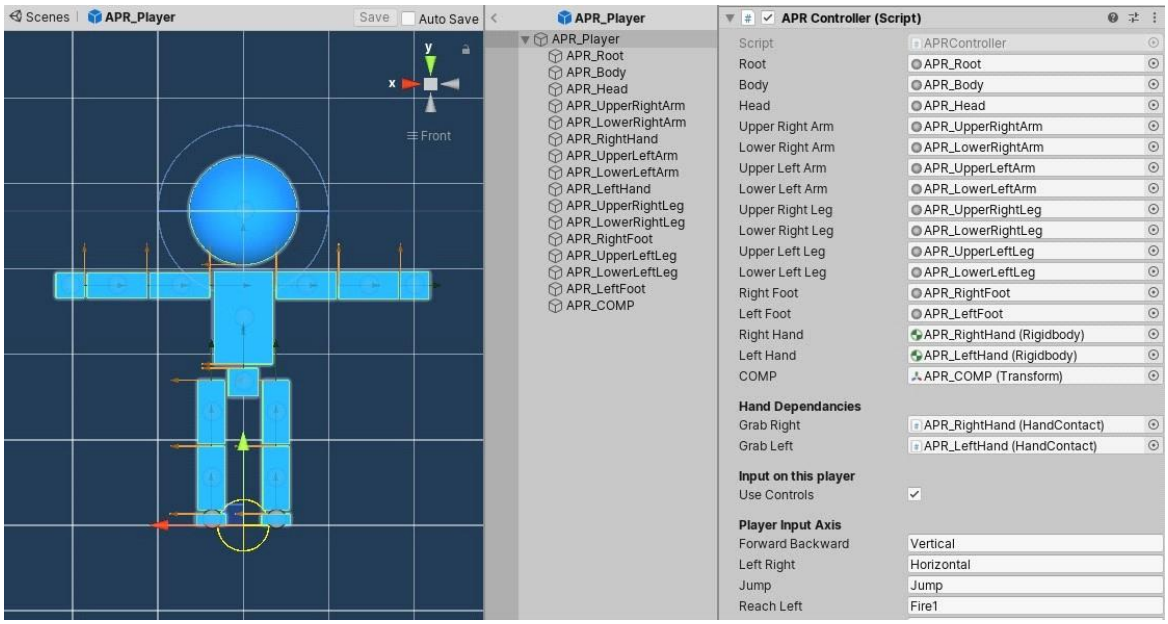
So with a better understanding of the foundation, let's take a look at how this method can be applied to a character in order to achieve a player system like the popular Gang Beasts game.

The principles of the duck are the same for player characters, with additional bones, joints and input script along with joint drive manipulation.

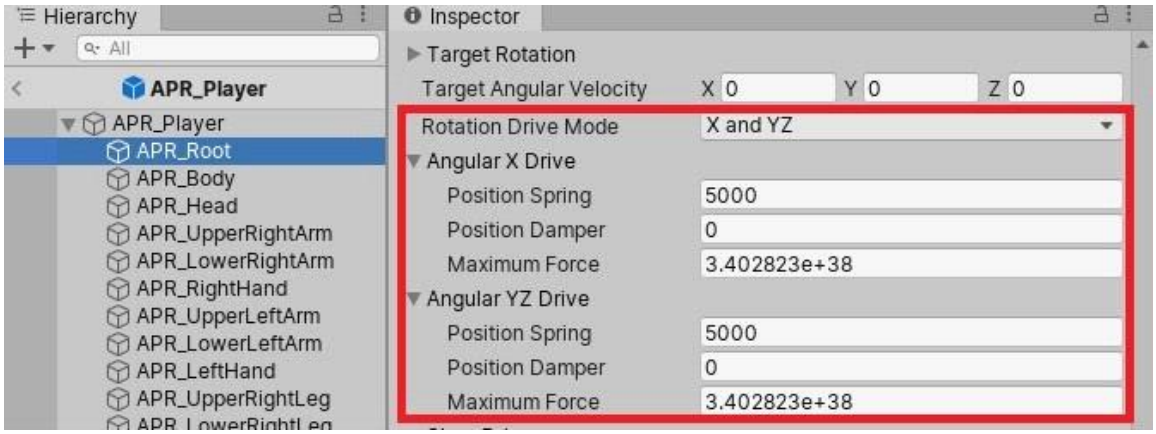
We'll be looking at the box model template, which can either be disected and build from

scratch onto a new model or simply be used for binding the existing box model objects with components to your model's bones using the APR Binder tool.

The character parts consists of A rigidbody, configurable joint and A collider component, the container has an APRController script attached, just like the duck example, we use this script to control the root bone called "APR_Root"
(A Free Non-restricted Configurable Joint)



The root bone joint drive "APR_Root"

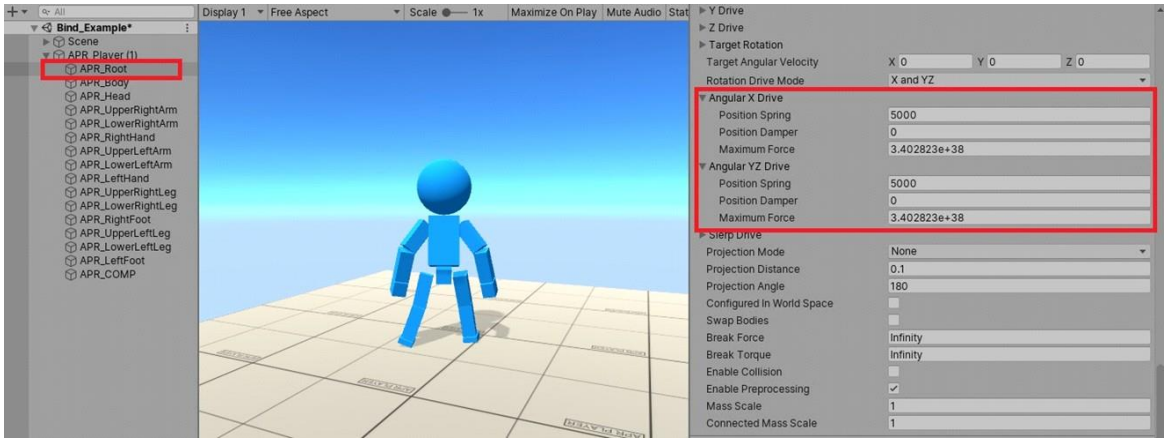


With characters, we can use these joint drives, the strength and target rotation by manipulating them in script for behavior purposes like losing balance by setting the root joint drive to zero or physically animate an arm by setting the target rotation.

Inside the APRController script I'm casting a raycast down from the "APR_Root" (referenced as APR_Parts[0]), by checking the raycast, whether or not it hits, if not we call a function that sets the joint drive to be weak enough for the root joint drive to no longer support the weight of the player's combined rigidbodies, the opposite when the raycast returns true.

```
308     //---Ground Check---//
309     ////////////////////
310     void GroundCheck()
311     {
312         Ray ray = new Ray (APR_Parts[0].transform.position, -APR_Parts[0].transform.up);
313         RaycastHit hit;
314
315         //Balance when ground is detected
316         if (Physics.Raycast(ray, out hit, balanceHeight, 1 << LayerMask.NameToLayer("Ground")) && !inAir && !isJumping && !reachRightAxisUsed && !reachLeftAxisUsed)
317         {
318             if(!balanced && APR_Parts[0].GetComponent<Rigidbody>().velocity.magnitude < 1f)
319             {
320                 if(autoGetUpWhenPossible)
321                 {
322                     balanced = true;
323                 }
324             }
325         }
326
327         //Fall over when ground is not detected
328         else if(!Physics.Raycast(ray, out hit, balanceHeight, 1 << LayerMask.NameToLayer("Ground")))
329         {
330             if(balanced)
331             {
332                 balanced = false;
333             }
334         }
335
336
337         //Balance on/off
338         if(balanced && isRagdoll)
339         {
340             DeactivateRagdoll();
341         }
342         else if(!balanced && !isRagdoll)
343         {
344             ActivateRagdoll();
345         }
346     }
```

Strong joint drive for balancing

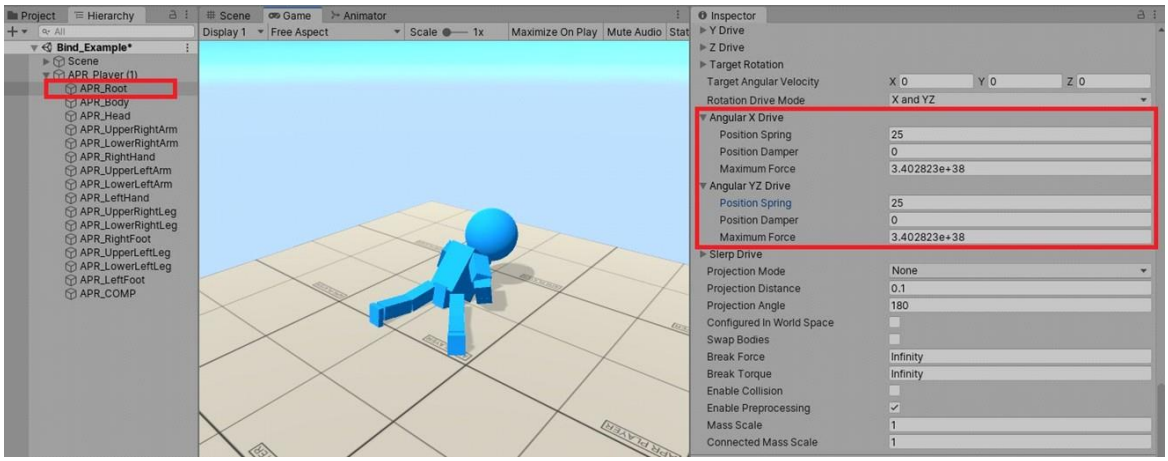


Setting root joint drive to be weak when raycast returns false

```
1041        //---Activate Ragdoll---//
1042        ////////////////////////
1043        public void ActivateRagdoll()
1044        {
1045            isRagdoll = true;
1046            balanced = false;
1047
1048            //Root
1049            APR_Parts[0].GetComponent<ConfigurableJoint>().angularXDrive = DriveOff;
1050            APR_Parts[0].GetComponent<ConfigurableJoint>().angularYZDrive = DriveOff;
```

I am using a low value such as 25 instead of zero to keep the player from tilting on its head when jumping, by jumping the limited raycast length loses contact with the ground causing the player to "lose balance"

(weak joint drive, unable to reach its upright target rotation)



For physically animating a character, we can set the joint drive high enough to enable us to control it by using the joint's target rotation.

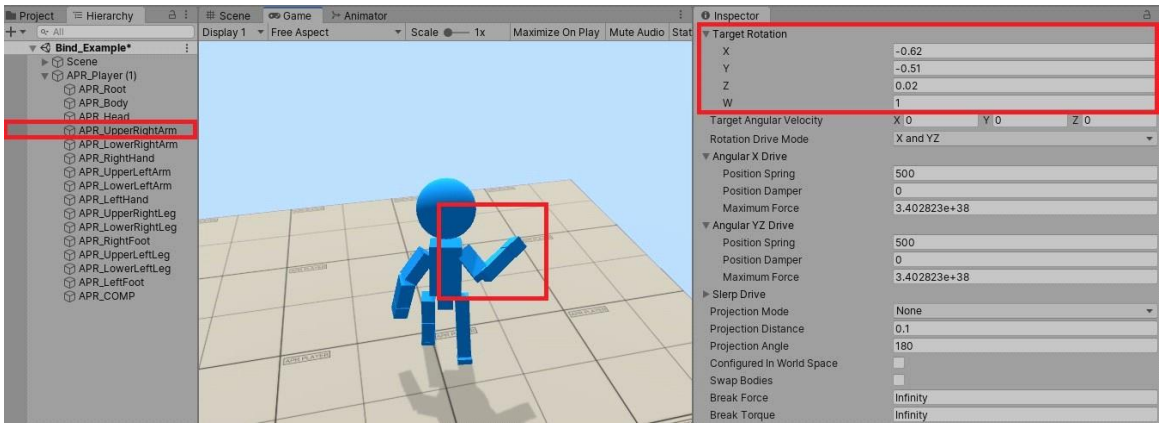
Here I've added A punching animation by mocking the arm when input is down then extend upon release, you could gradually increase the joint drive value over time for more precise control like I did with walking simulation but in this case I used a 2 point approach, because the joint drive will correct the current rotation to match the target rotation using the joint drive strength, so no need to fill in the gap from point A to point B as the drive will do so itself.

I set a value when the input is down

Notice APR_Parts[3] (The right upper arm)

```
811     //---Player Punch---//
812     ////////////////////
813     void PlayerPunch()
814     {
815
816         //punch right
817         if(!punchingRight && Input.GetKey(punchRight))
818         {
819             punchingRight= true;
820
821             //Right hand punch pull back pose
822             APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, -0.15f, 0, 1);
823             APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.62f, -0.51f, 0.02f, 1);
824             APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 1.31f, 0.5f, -0.5f, 1);
825         }
826
827         if(punchingRight && !Input.GetKey(punchRight))
828         {
829             punchingRight = false;
830
831             //Right hand punch release pose
832             APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, 0.15f, 0, 1);
833             APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.74f, 0.04f, 0f, 1);
834             APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.2f, 0, 0, 1);
835
836             //Right hand punch force
837             RightHand.AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse);
838
839             APR_Parts[1].GetComponent<Rigidbody>().AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse);
840
841             StartCoroutine(DelayCoroutine());
842             IEnumerator DelayCoroutine()
843             {
844                 yield return new WaitForSeconds(0.3f);
845                 if(!Input.GetKey(punchRight))
846                 {
847                     APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = UpperRightArmTarget;
848                     APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = LowerRightArmTarget;
849                 }
850             }
851         }
852
```

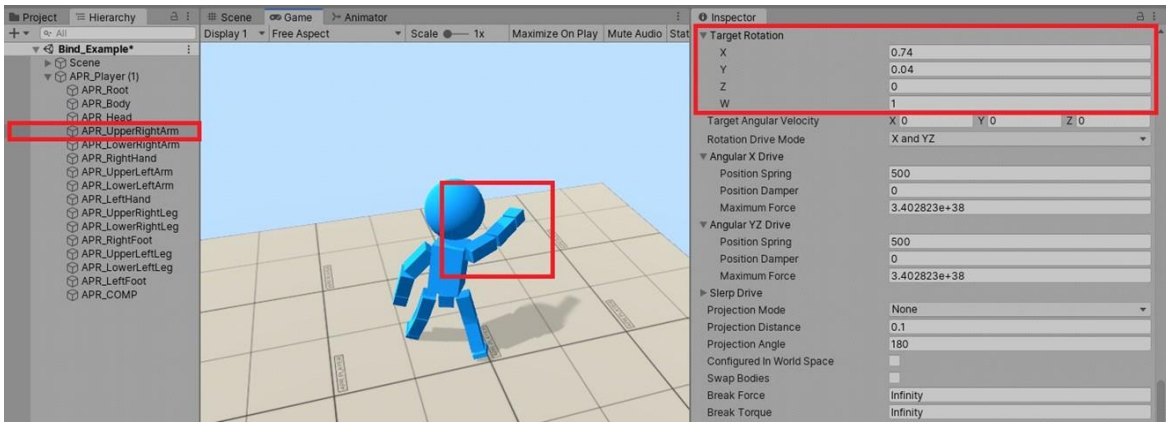
The joint then tries to match the value asked for while taking physics into account



On input release I give the target rotation a new value

```
//---Player Punch---//
////////////////////
void PlayerPunch()
{

    //punch right
    if(!punchingRight && Input.GetKey(punchRight))
    {
        punchingRight= true;

        //Right hand punch pull back pose
        APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, -0.15f, 0, 1);
        APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.62f, -0.51f, 0.02f, 1);
        APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 1.31f, 0.5f, -0.5f, 1);
    }

    if(punchingRight && !Input.GetKey(punchRight))
    {
        punchingRight = false;

        //Right hand punch release pose
        APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, 0.15f, 0, 1);
        APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.74f, 0.04f, 0f, 1);
        APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.2f, 0, 0, 1);

        //Right hand punch force
        RightHand.AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse);

        APR_Parts[1].GetComponent<Rigidbody>().AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse)

        StartCoroutine(DelayCoroutine());
        IEnumerator DelayCoroutine()
        {
            yield return new WaitForSeconds(0.3f);
            if(!Input.GetKey(punchRight))
            {
                APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = UpperRightArmTarget;
                APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = LowerRightArmTarget;
            }
        }
    }
```

The joint drive tries to correct to the new given value



After this cycle I am simply re-setting the target rotation value to the initial idle state, A value that was stored.

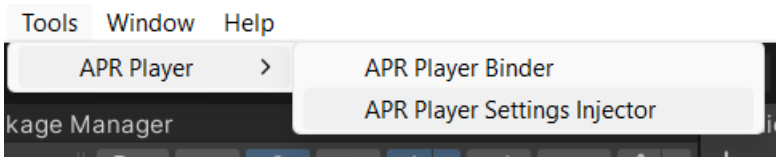
```
//---Player Punch---//
////////////////////
void PlayerPunch()
{

    //punch right
    if(!punchingRight && Input.GetKey(punchRight))
    {
        punchingRight= true;

        //Right hand punch pull back pose
        APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, -0.15f, 0, 1);
        APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.62f, -0.51f, 0.02f, 1);
        APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 1.31f, 0.5f, -0.5f, 1);
    }

    if(punchingRight && !Input.GetKey(punchRight))
    {
        punchingRight = false;

        //Right hand punch release pose
        APR_Parts[1].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( -0.15f, 0.15f, 0, 1);
        APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.74f, 0.04f, 0f, 1);
        APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = new Quaternion( 0.2f, 0, 0, 1);

        //Right hand punch force
        RightHand.AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse);

        APR_Parts[1].GetComponent<Rigidbody>().AddForce(APR_Parts[0].transform.forward * punchForce, ForceMode.Impulse);

        StartCoroutine(DelayCoroutine());
        IEnumerator DelayCoroutine()
        {
            yield return new WaitForSeconds(0.3f);
            if(!Input.GetKey(punchRight))
            {
                APR_Parts[3].GetComponent<ConfigurableJoint>().targetRotation = UpperRightArmTarget;
                APR_Parts[4].GetComponent<ConfigurableJoint>().targetRotation = LowerRightArmTarget;
            }
        }
    }
```

See animated GIF of this effect here:

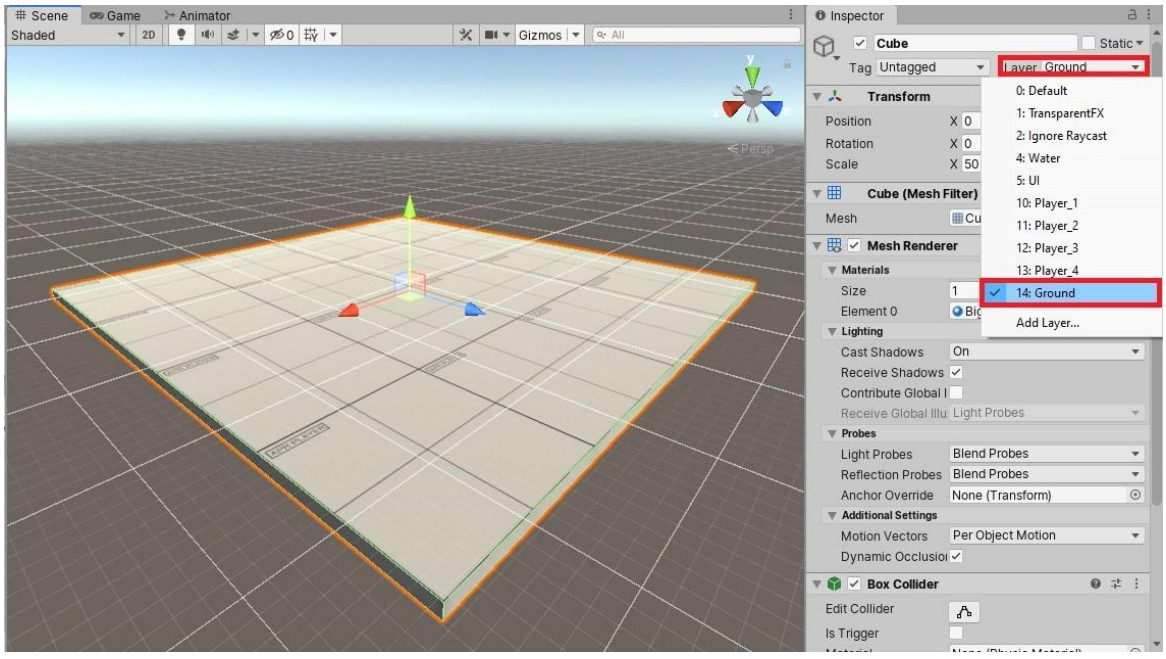
https://drive.google.com/file/d/1lsw3xoXaTFUN91fBM4hVNa48Zzresrrw/view?usp=sharing

Setting Up A New Project

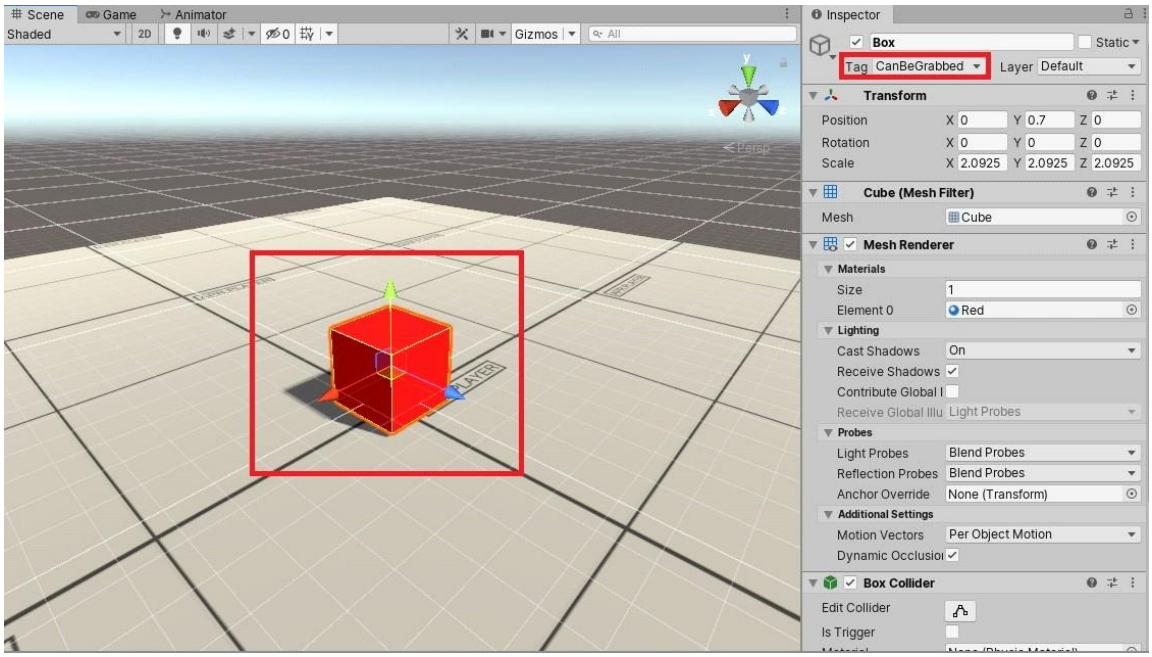
After importing the APR Player package into a new project, the first thing to do is setting up the correct project settings, if you are importing from the Unity asset store then the project settings should be included, if you are importing the package without these settings already applied you can simply inject them with the APR Settings injector tool.



Once you've got the project ready, you can setup a functional scene for the APR Player by creating a floor object and adding it to the "Ground" layer.



An object that can be grabbed by the APR Player, add a rigidbody and then tag it as "CanBeGrabbed"

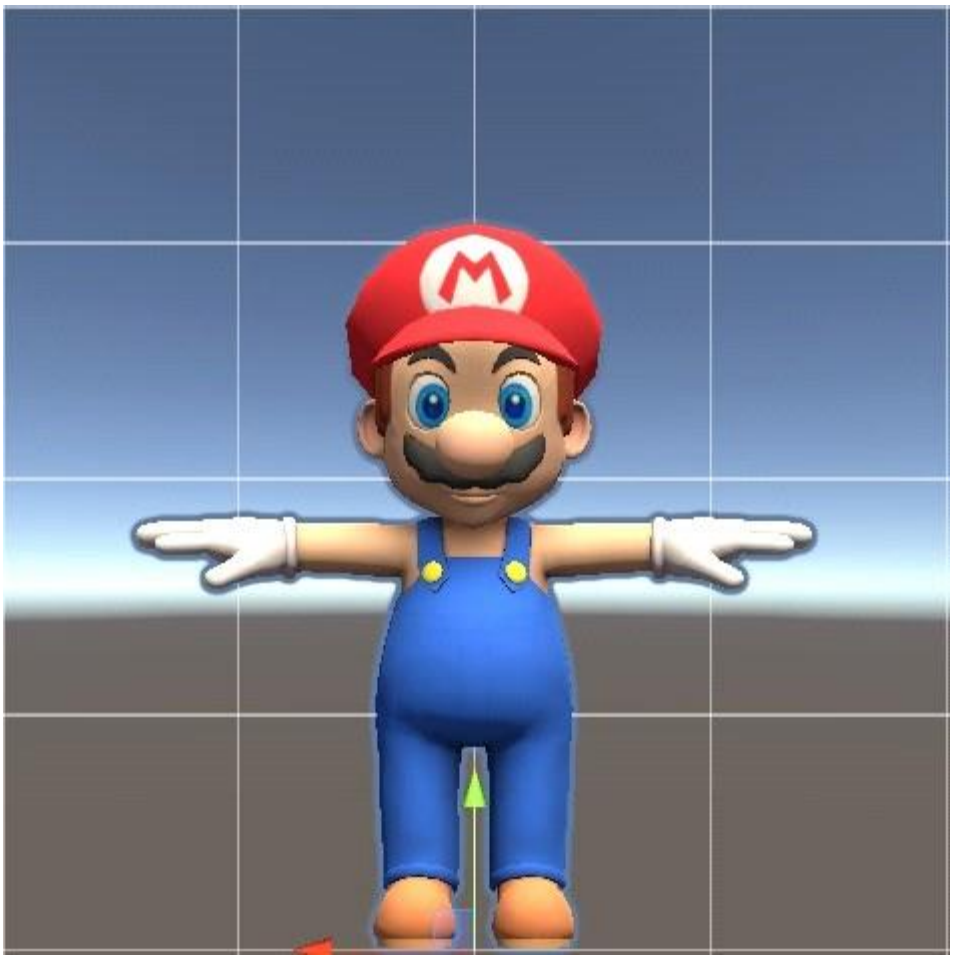


Binding New Characters

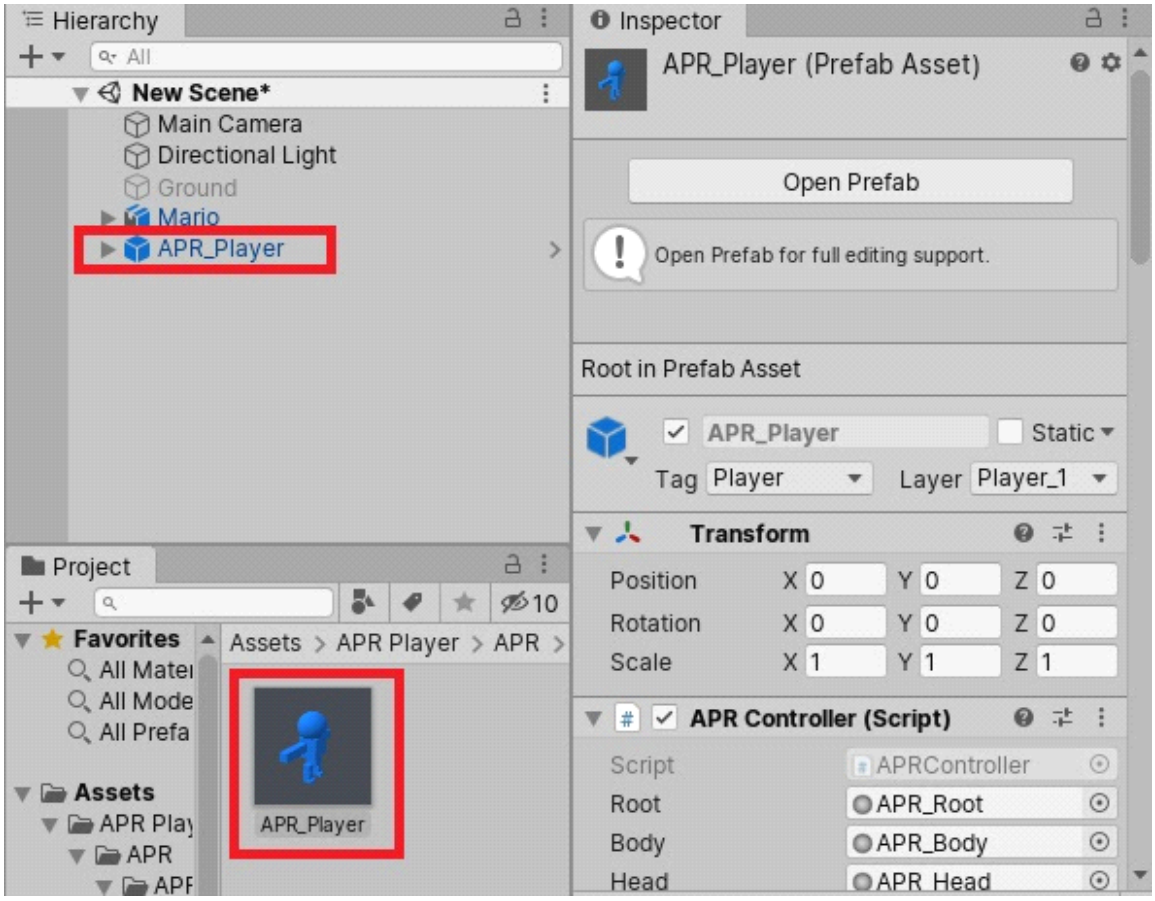
Binding new characters make the use of the already pre-made active physics ragdoll box model player, to setup your own model you'll need a rigged character with A basic humanoid bone tree.

Root(Pelvis), Body, Head, Upper Arms, Lower Arms, Hands, Upper Legs, Lower Legs and Feet bones.

Create A new scene, drag your rigged model into the scene hierarchy

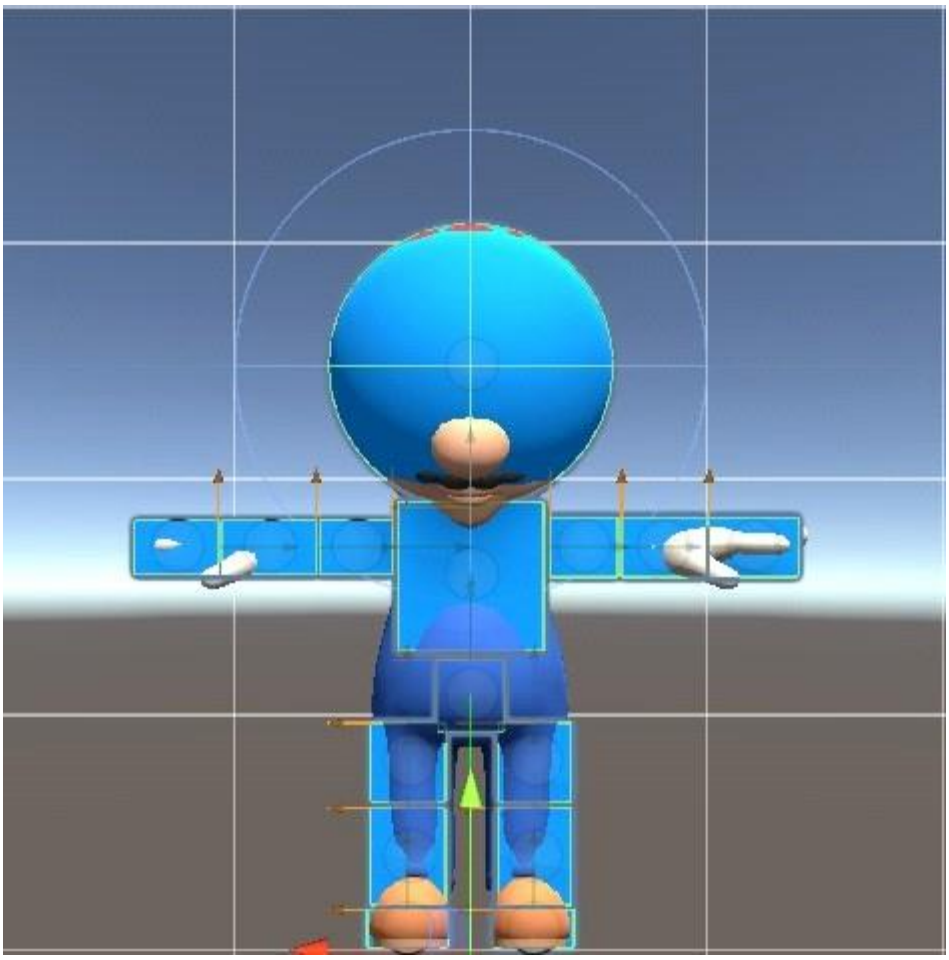


Add one of the pre-made box model prefabs to the scene hierarchy

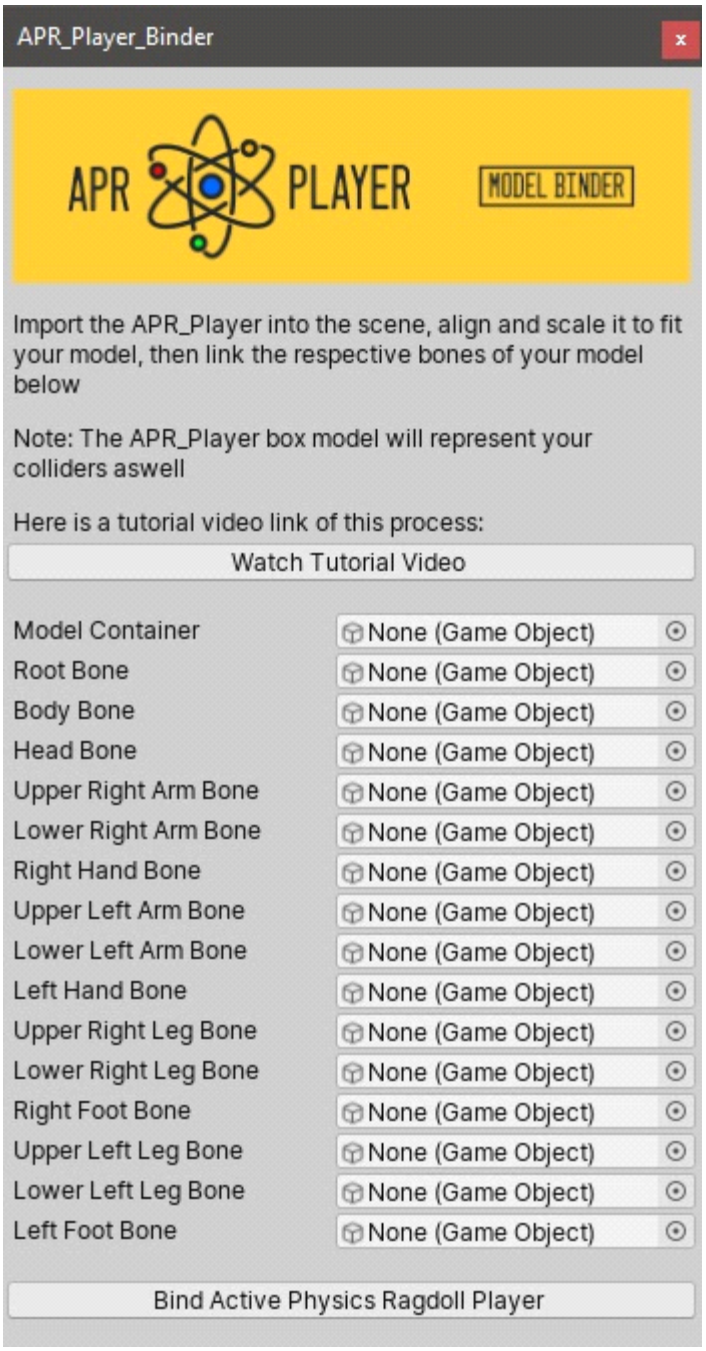


To do as little property adjustments as possible you can first scale your entire rigged model to relatively match the box model, then position and scale the blue boxes to match your model's limbs.

Think of the blue boxes as your model's colliders, because they will be.



When you're satisfied with the alignment, open the binder tool from the top window.

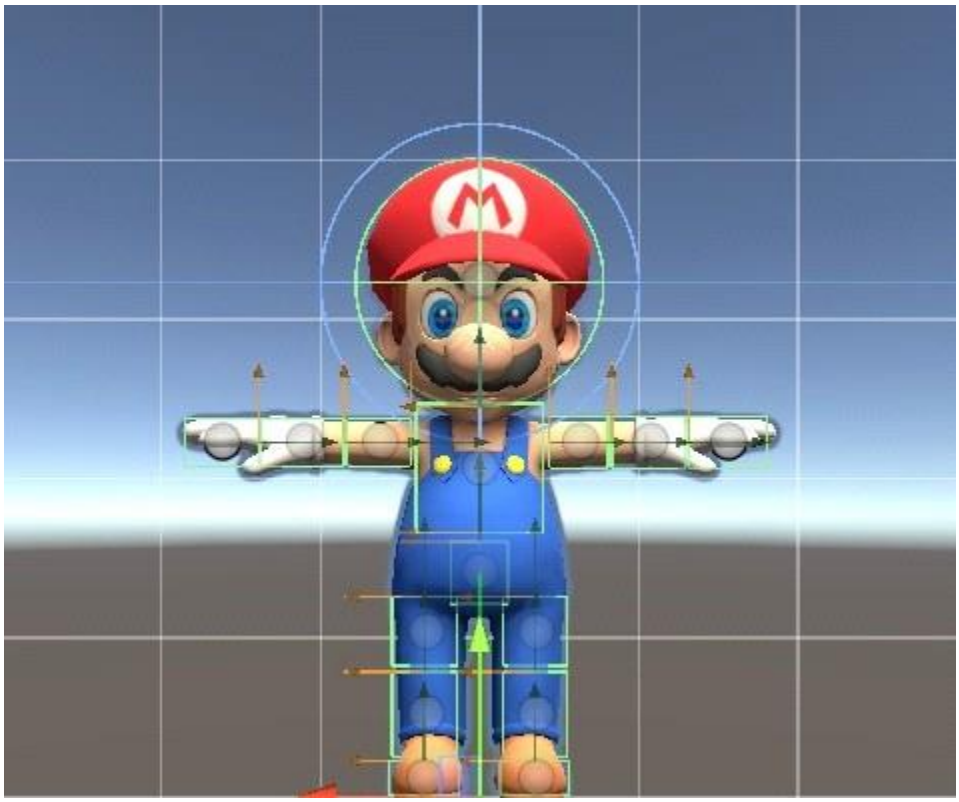


After assigning your model's bones to the fields, press the "Bind Active Physics Ragdoll Player" button.

You should then have the box model objects with components integrated into your rigged

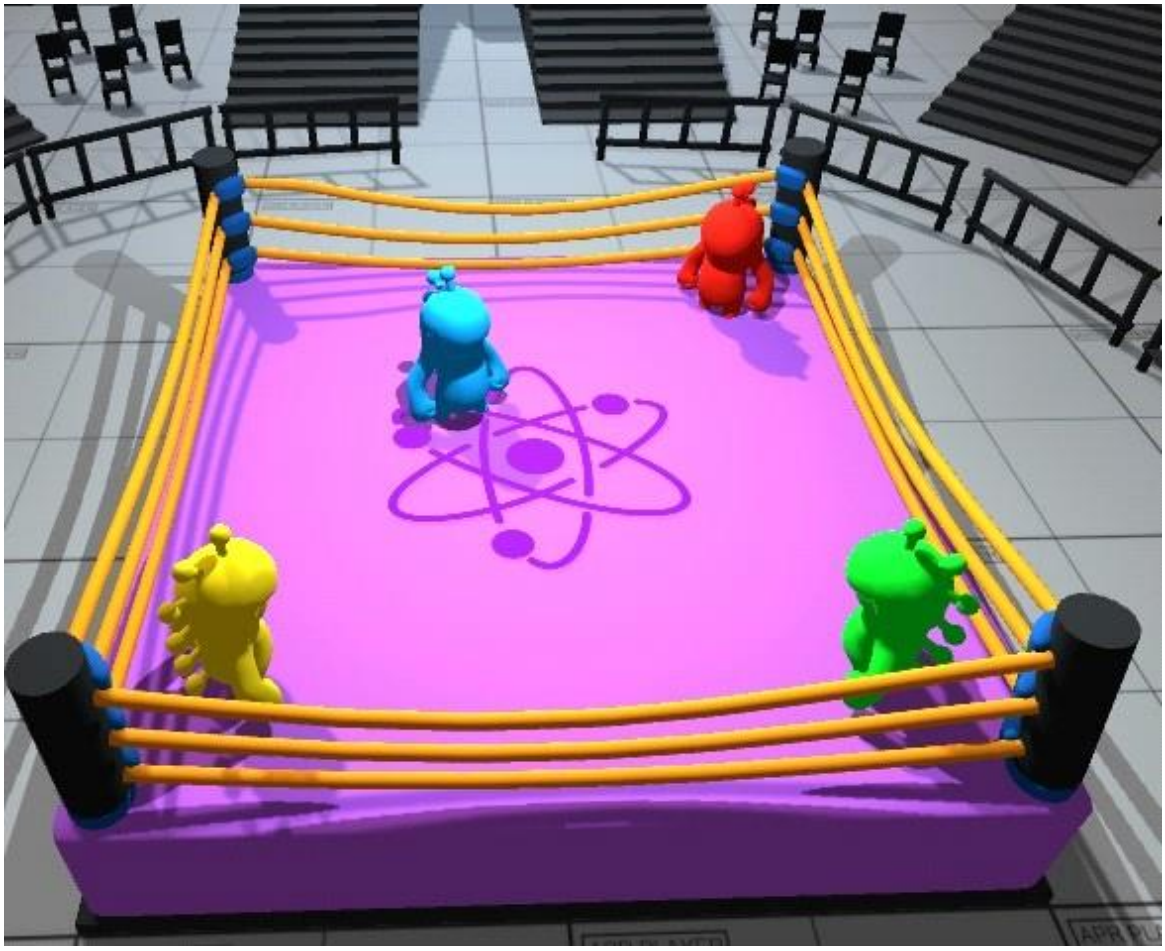
model's bones tree and have a new functional APR Player. If your character falls over, never gets back up or drag its feet when walking then its because your model is significantly different from the pre-made template, for example it has very long legs.

If so, adjust the properties in the APRController script, balance height (The raycast length to detect ground), stepping height (How high the legs are being picked up off the ground when moving) etc.

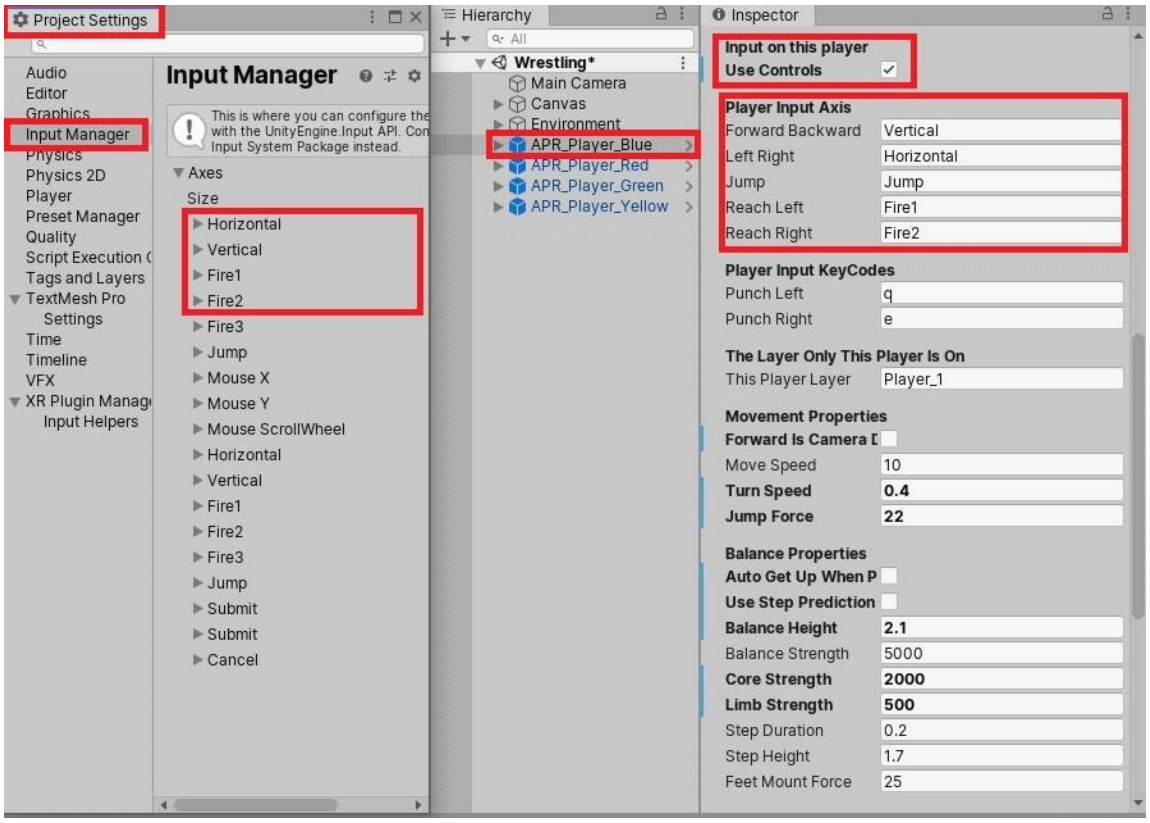


Local Multi Player

The APR Player can easily be used for couch party local multi player games. The controller script uses the input axis manager strings which means you can add multiple new axes like the use of 2-4 gamepads and then type those names into the APRController properties for each player to control them with different gamepad controllers.



The blue character has been setup with Unity's default keyboard axes, in the input manager you'll want to create your own axis with the options for keyboard or joysticks along with the keys. Create axes for each player and then assign the strings into their APRController property.


Project Settings
Audio
Editor
Graphics
Input Manager
Physics
Physics 2D
Player
Preset Manager
Quality
Script Execution
Tags and Layers
TextMesh Pro
Settings
Time
Timeline
VFX
XR Plugin Manag
Input Helpers
Input Manager
This is where you can configure the
with the UnityEngine.Input API. Con
Input System Package instead.
Axes
Size
Horizontal
Vertical
Fire1
Fire2
Fire3
Jump
Mouse X
Mouse Y
Mouse ScrollWheel
Horizontal
Vertical
Fire1
Fire2
Fire3
Jump
Submit
Submit
Cancel
Hierarchy
All
Wrestling*
Main Camera
Canvas
Environment
APR_Player_Blue
APR_Player_Red
APR_Player_Green
APR_Player_Yellow
Inspector
Input on this player
Use Controls
Player Input Axis
Forward Backward
Vertical
Left Right
Horizontal
Jump
Jump
Reach Left
Fire1
Reach Right
Fire2
Player Input KeyCodes
Punch Left
q
Punch Right
e
The Layer Only This Player Is On
This Player Layer
Player_1
Movement Properties
Forward Is Camera D
Move Speed
10
Turn Speed
0.4
Jump Force
22
Balance Properties
Auto Get Up When P
Use Step Prediction
Balance Height
2.1
Balance Strength
5000
Core Strength
2000
Limb Strength
500
Step Duration
0.2
Step Height
1.7
Feet Mount Force
25