INTENSE TPS

DOCUMENTATION

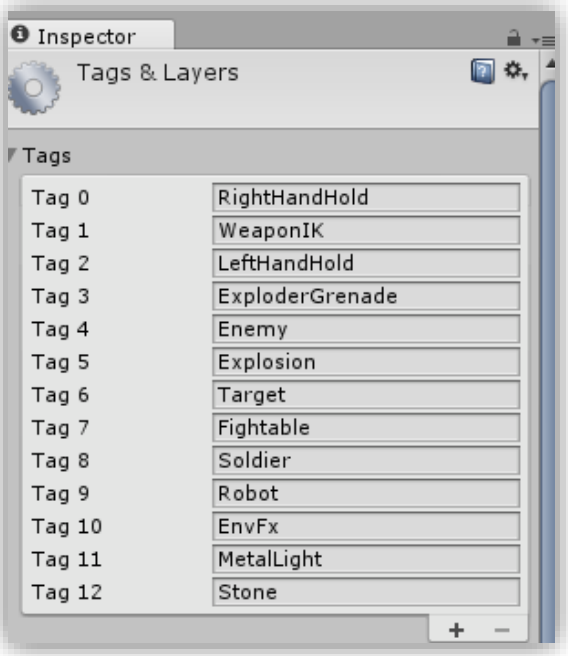
Contact mail : elegan@outlook.com.tr

Version : 1.1

1. Importing project and using prefabs.

a. Layers and tags

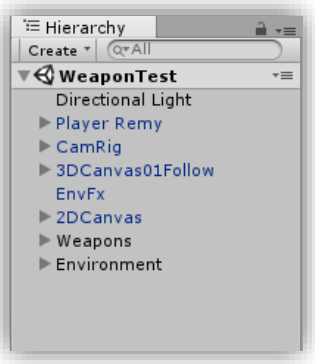
In this project, layers and tags are used for various purposes. Since this is a complete project, when the project is imported, all tags



and layers should also be imported automatically. Make sure those tags and layers are available. Restart unity after importing if necessary.

b. Scene management

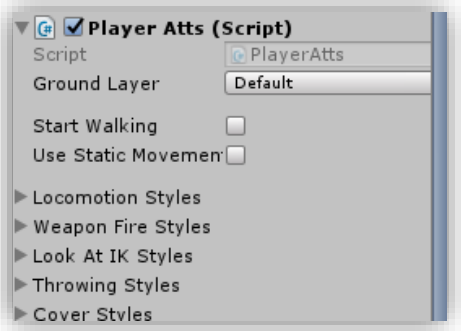
Currently, there are 3 example scenes available in the project. Two of them includes player. As an example let's take a look at what gameobjects are available in the hierarchy of ***"WeaponTest"*** scene.



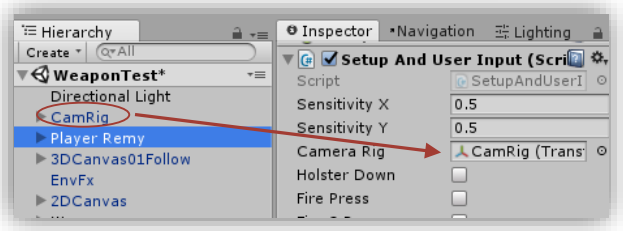
There are three prefabs : Player Remy, CamRig, 3Dcanvas01Follow, EnvFx, 2Dcanvas.

1: Camrig and EnvFx need to be in the scene, otherwise you will get errors (by player), and in order for 3Dcanvas01Follow and 2Dcanvas to work, one player need to be available in the scene.

2: There should be only one player prefab ***active*** in the scene. So if you work with more than one player prefabs deactivate others before pressing play.

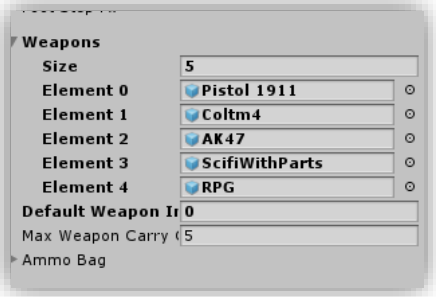


3: You have created a scene you dragged a player prefab but if the player is refusing to land to ground you created, that means the ground (or ground collider) layer is not available in the player script's ground layer.



4: You need to show ***CamRig*** to player's ***SetUpAndUserInput*** component.

5: Most of the adjustments of player can be done using ***PlayerAtts*** component of player prefab, including assigning new weapons. The weapons that assigned to player, are must be available in the scene. They are not prefabs, so make sure to drop them to scene before assigning. (However, throwables that are assigned to player, are prefabs)



6: Keep in mind before importing, that all your project settings will be overriden when you import this project, since this is a complete project.

2. Controls

All control keys & control logic can be found in ***SetUpAndUserInput*** script. Project's default input settings are not modified for generality (Except for keyboard Up/Down/Left/Right arrows).

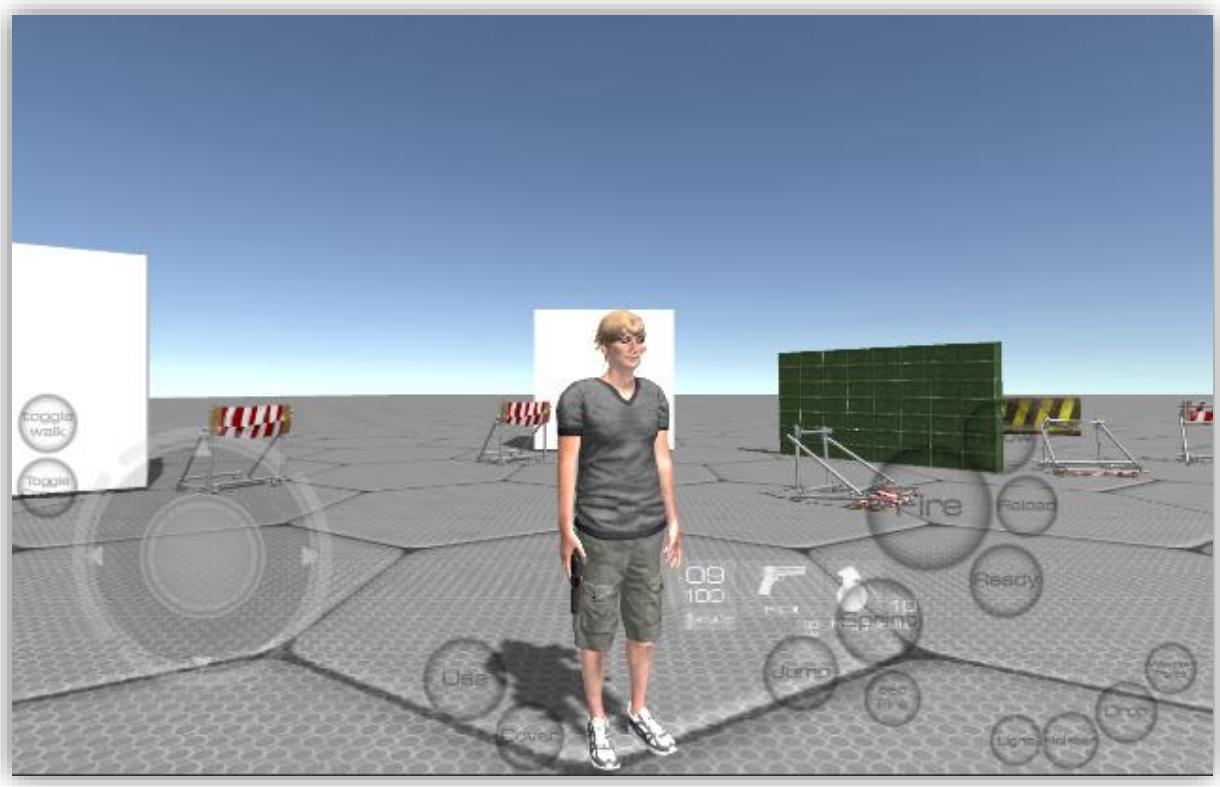
Out of the box supperted control platforms are keyboard/mouse and touch screen.

a. Keyboard and mouse

1: Keyboard and mouse

a. *Locomotion: W/A/S/D*
b. *Sprint: Shift*
c. *Collect: E*
d. *Jump: Space*
e. *Pullout/Holster weapon: F*
f. *Throw grenade: G*
g. *Drop weapon: H*
h. *Crouch/Stand: C*
i. *Toggle Walk/Run: LeftAlt*
j. *Fire: Mouse0*
k. *Sight/Cover Aim: Mouse1*
l. *Hipfire Aim: Mouse0*
m. *Reload: R*
n. *SecondaryFire: Mouse2*
o. *Camera/Player rotation: Mouse*
p. *Modify weapon: V*
q. *Weapon Flashlight: T*
r. *Change weapon/throwable: Keyboard arrows*

b. Touchscreen



In order to use touch screen controls, ***MobileControlRig*** prefab must be available in the hierarchy and ***Mobile Input*** must be enabled from Unity editor menubar. Mobile control rig is capable of many things that some mobile third person games are using like firing while rotating camera, swiping to change throwable/weapon.

Keep in mind that mobile controls will not be active until you switch to a supported platform and some of the features will only be available if you use Unity Remote application or after installing the exported application to a device.

3. Change Character

If your character is humanoid, you can easily set it as your new player. You can check if your character is compatible by clicking configure button in inspector avatar definition. If all the bones are green it means that it is compatible to use with humanoid animations.

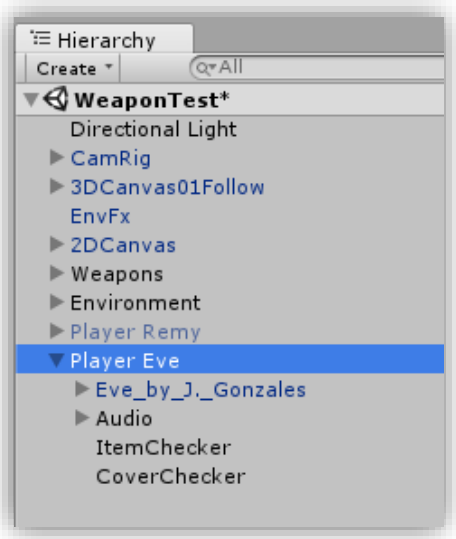
a. Quick character setup

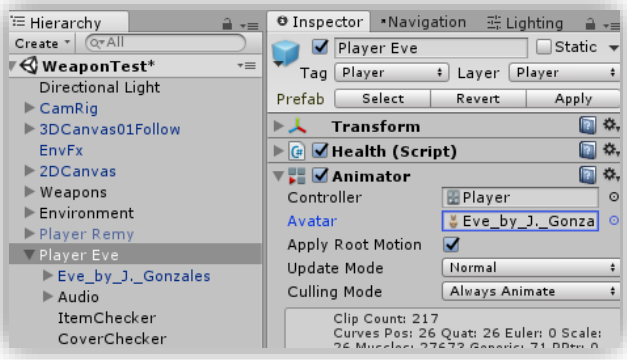
1- Make sure you have imported character as humanoid and import options are matches this picture.

2- Start a new scene or open ***WeaponTest*** scene like I am going to do. Drop one of the player prefabs into the scene. Duplicate it and rename the gameobject's name. Then drag your new character model into the scene. Disable old prefab as we will not use it.

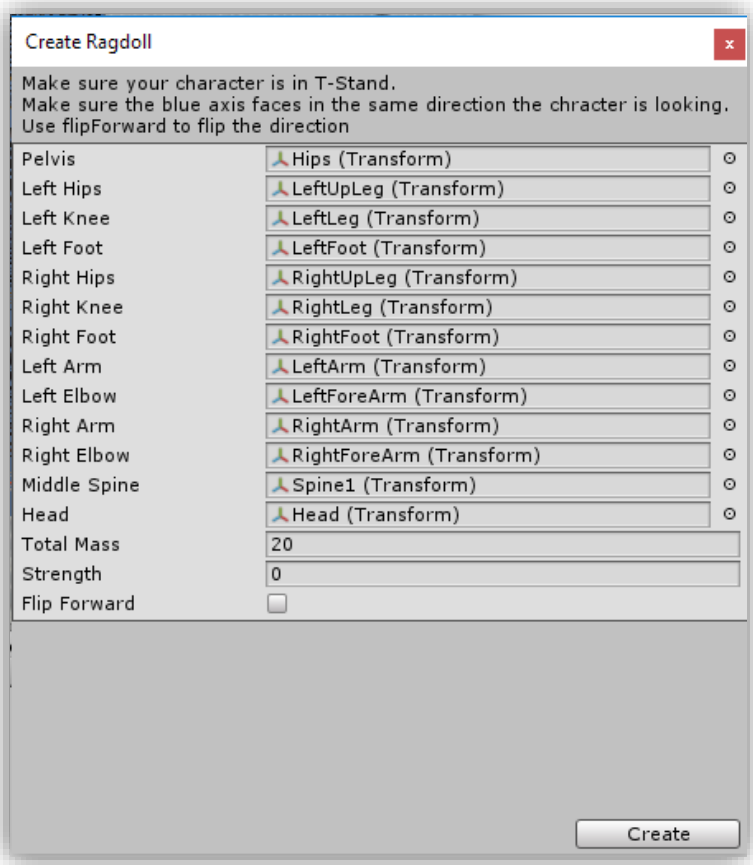
When I first imported the character you see in this picture, she was too small then my player Remy character, so I changed my new character model's scale factor so that their heights match.

3- Drop your character model to duplicated player prefab. Reset model's position and rotation. (Remove model's ***Animator*** component as we will use Player's ***Animator***) Drop your character model's ***Avatar*** to Player's ***Animator*** component.

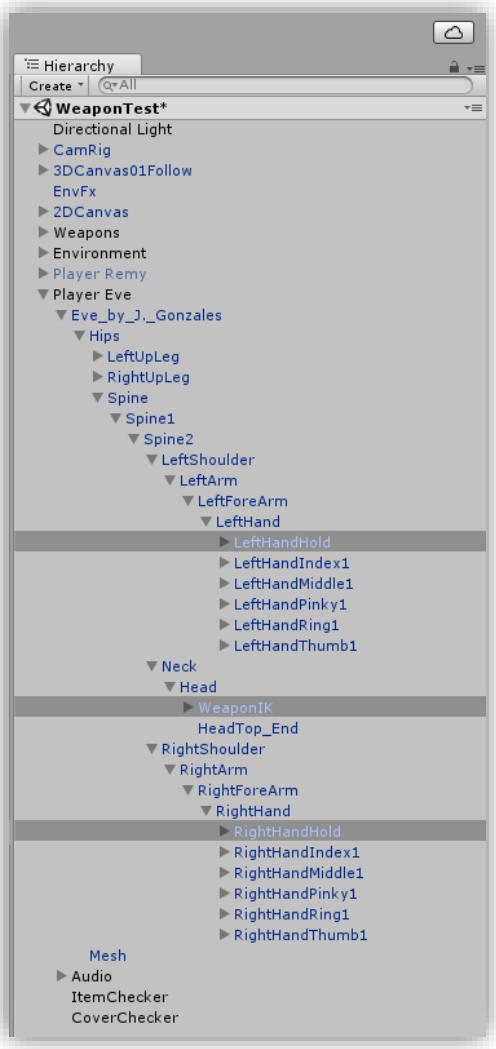




4- Let's create a ***Ragdoll*** for the character.
From the unity editor menubar, open ***GameObject/3D Object/Ragdoll*** window. Then drop needed bones into the fields. Hit create to finish ragdoll.



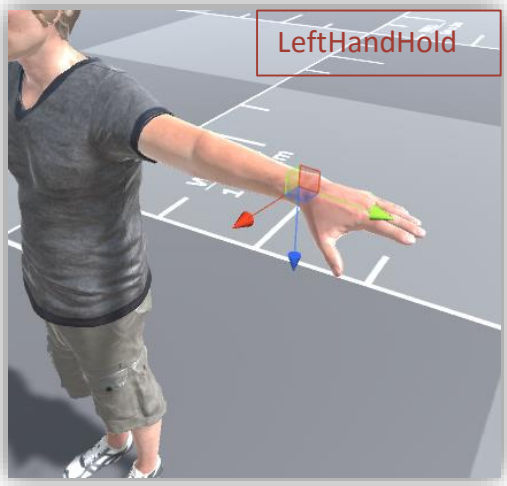
5- In your project tab, navigate to ***IntenseTPS/Prefabs/Player/SettingUp*** folder. You will see three prefabs that will be used to rig character. Using hierarchy:

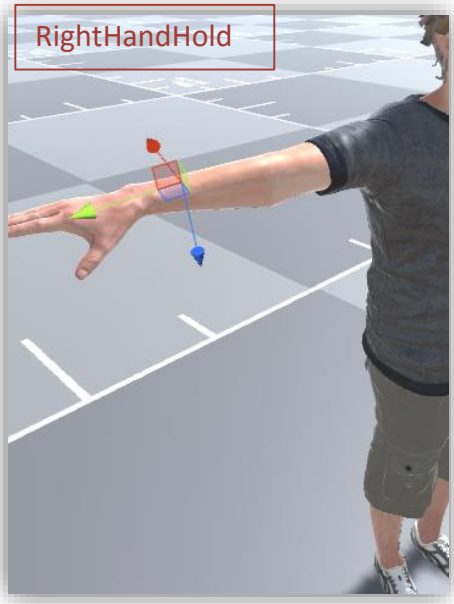


   a. Drop ***LeftHandHold*** prefab from project tab to character's left hand.
   b. Drop ***RightHandHold*** prefab from project tab to character's right hand.
   c. Drop ***WeaponIK*** prefab from project tab to character's Head.

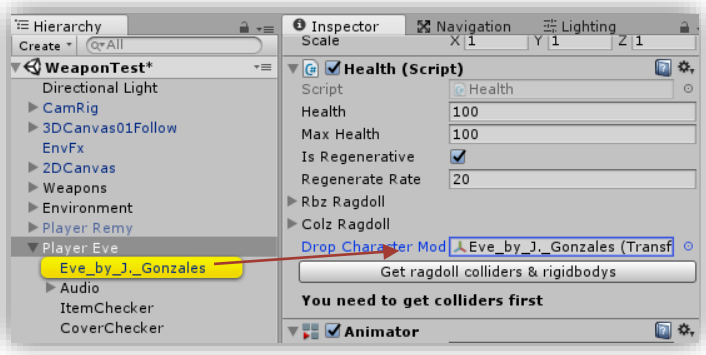
Then, reset position and rotation of these objects transform components.

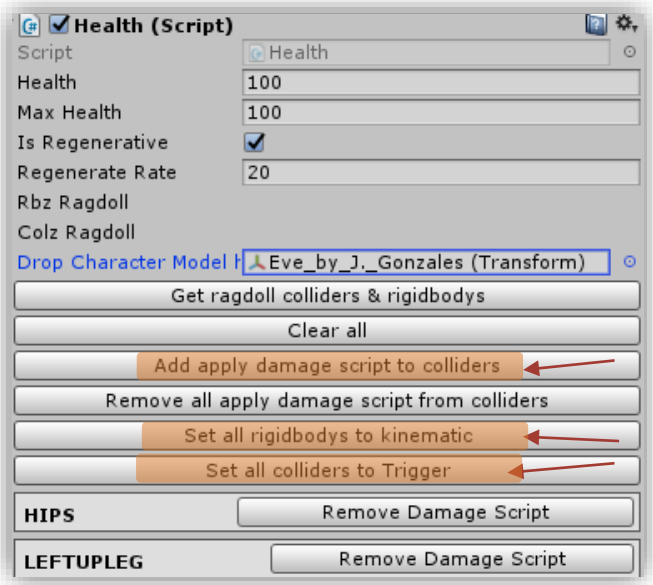
6- Set rotations of the objects you have dropped in the previous step, so that they match the ones you see below. Keep in mind that If your character skeleton is setup up to be used for humanoid, most probably you will use +/-90, +/-180 local rotations while completing this step...



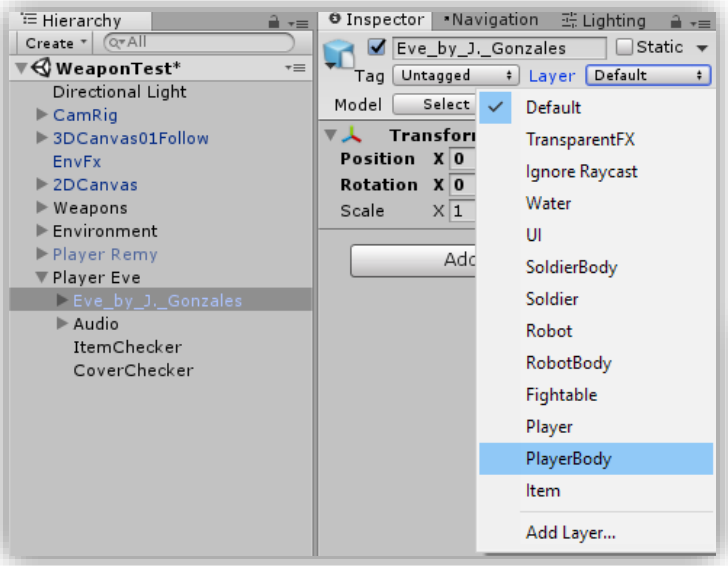


7- Select new player and from the inspector find the ***Health*** component. We will use this script to quickly disable all rigidbodies and ragdoll colliders. Drop your character's model or model's hips (which is child of player in this case) to ***Drop Character Model hips here*** field of ***Health*** component. Click ***Get ragdoll colliders & rigidbodys*** button.

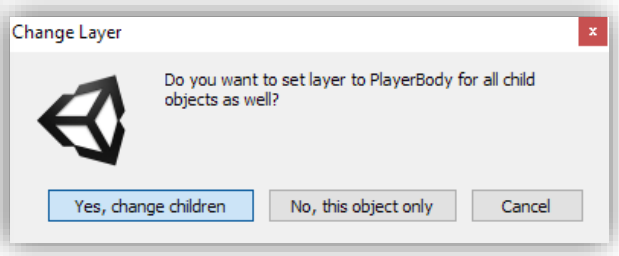




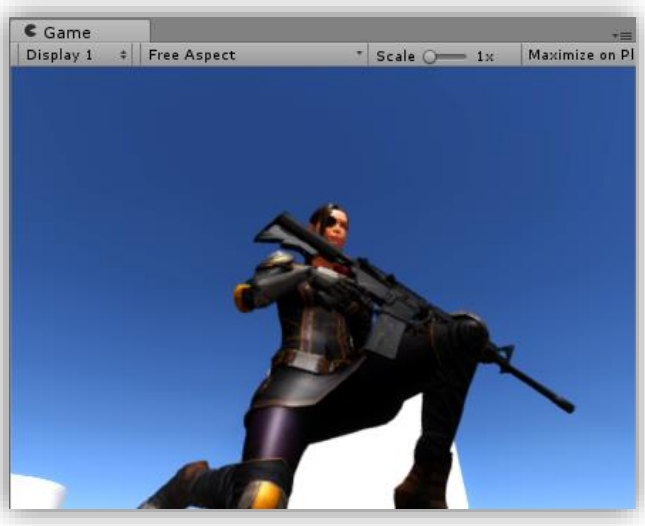
8- Click the buttons you see in the left picture. Order is not important.



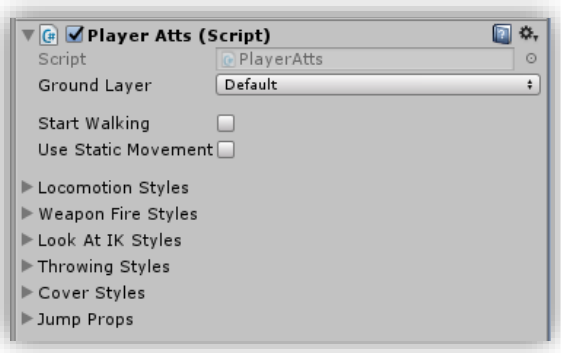
9- Select the new character model of player and set its layer to ***PlayerBody***, then hit Yes, change children button.



That is all you need do to rig up your model. Everything should work as expected now. For fine tuning weapon holding, you can read the ***Weapon System*** section.



4. Player Systems



In this project, player logic is split into many parts, I call those parts: player systems.

Almost every system has its serialized classes which holds fields (except some small systems) that can be modified using inspector as you can see in the picture right .

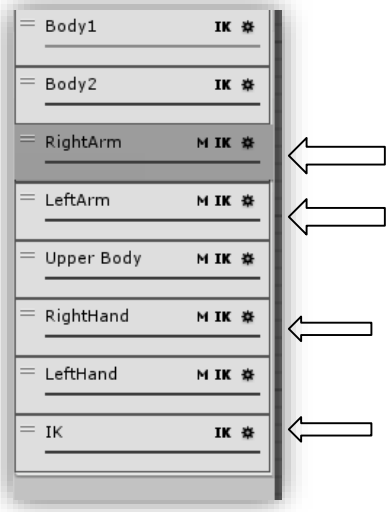
If you want to define new systems I strongly recommend reading coding section.

a. Weapon System

This system controls almost everything about guns. From pulling out weapon to firing, holstering etc.

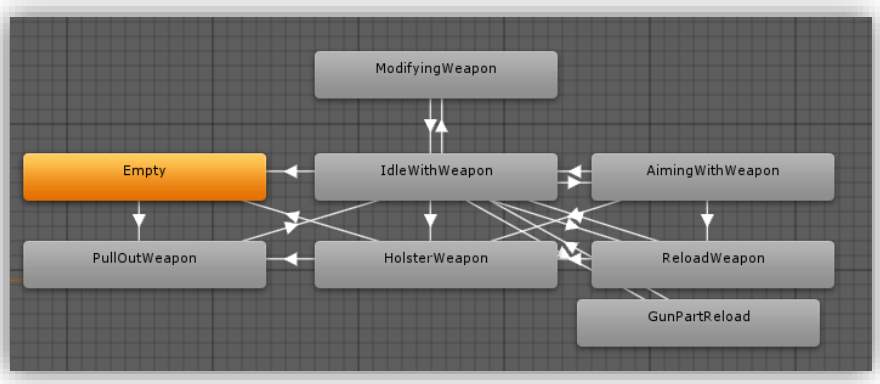
Almost all weapon types are supported by this system. Currently, there are 5 weapon prefabs available in the project, but system is not limited with them. After reading this section you will be able to create your own weapon.

a.1 Defining New Player Animations for New Weapons



Weapon System uses 4 layers in the Animator: ***RightArm***, ***LeftArm***, ***RightHand***, ***LeftHand***. The ***LeftArm*** layer imitates ***RightArm*** layer. The StateMachine named ***Weapon*** you see in the ***LeftArm*** layer is exact copy of the one you see in the ***RightArm***. So if you modify ***RightArm*** to define new animations for new weapons just copy and replace the ***Weapon*** statemachine of ***LeftArm*** after you finish.

Let's take a look at the animation blend trees that ***Weapon System*** is using (***RightArmLayer***).

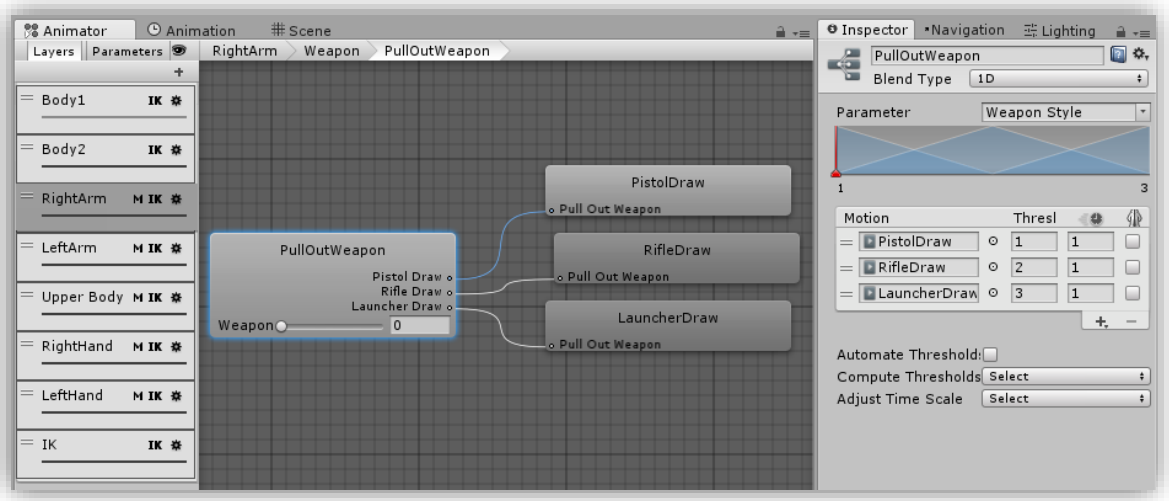


Important blend trees that player uses for every weapon :

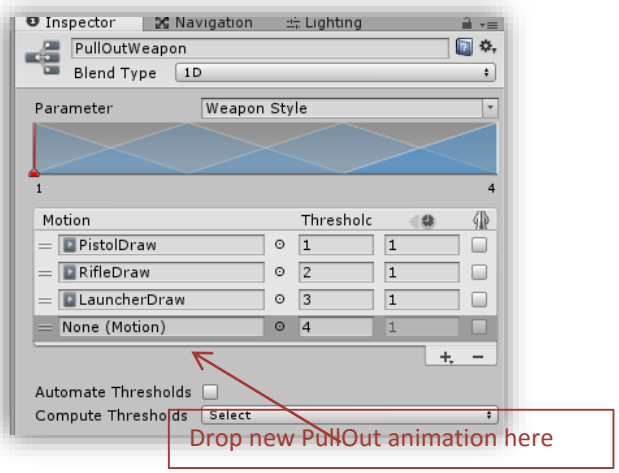
1- ***PullOutWeapon***
2- ***IdleWithWeapon***
3- ***ReloadWeapon***
4- ***AimingWithWeapon***

You will need to modify these blend trees like described below.

I will describe defining animations using ***PullOutWeapon*** (1) blend tree, because modifying other blend trees (2,3,4) are exactly the same.



If you look at the picture above, ***PullOutWeapon*** blend tree is using ***Weapon Style*** parameter to switch animations per weapon. Actually blending is not used here, that means ***Weapon Style*** parameter can't be 1.5 or 2.3 etc. Blend tree is used only for simplicity. You can see that exact numbers are entered in the ***Threshold*** fields of animations of the blend tree.



So let's say I have a weapon that I want player to use ***Weapon Style*** No:***4*** animations when holding it, I will only need to add another motion field to ***PullOut*** blend tree, set its Threshold to 4 and drop new animation there.

If you add all other animations and set a weapon's ***weaponStyle*** field of ***GunAtt*** component to the same number, player will use those animations when using that weapon.

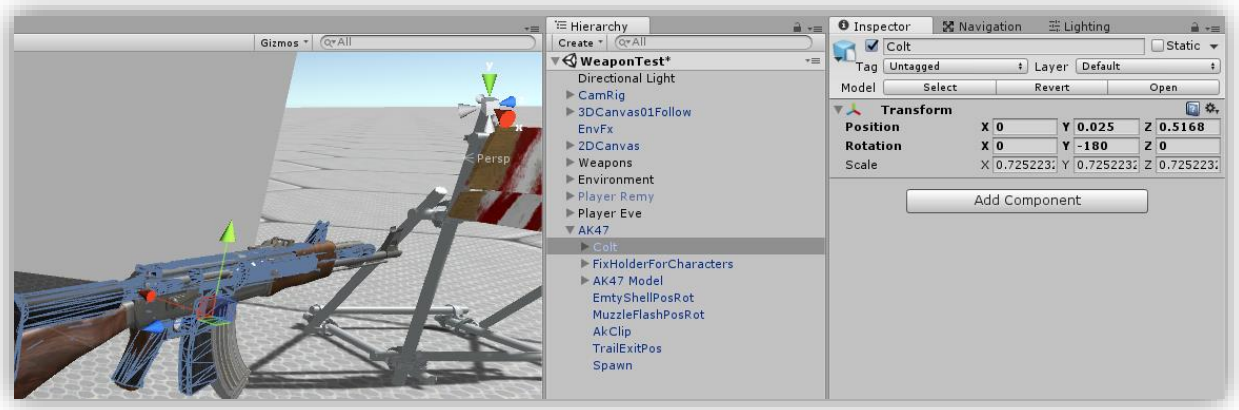
The other two animation/blend trees are used by modifiable weapons. If you want to play with them, modifying them is similar to the one explained.

a.2 Defining New Weapon

You will need a weapon model for this. After importing new weapon model, follow the instructions below.

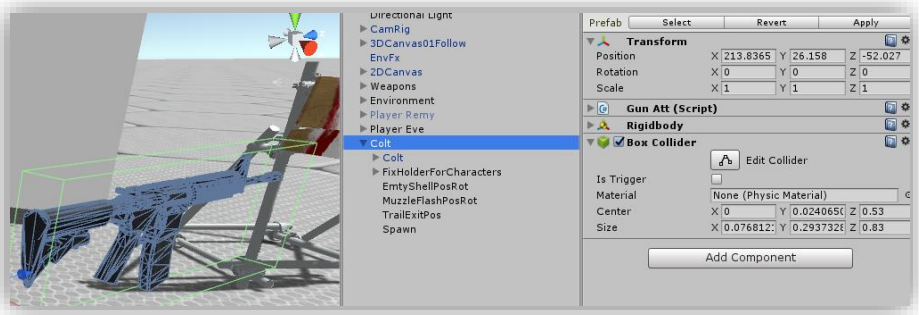
I will set up the weapon model named ***Colt*** which is available in the project, from stratch.

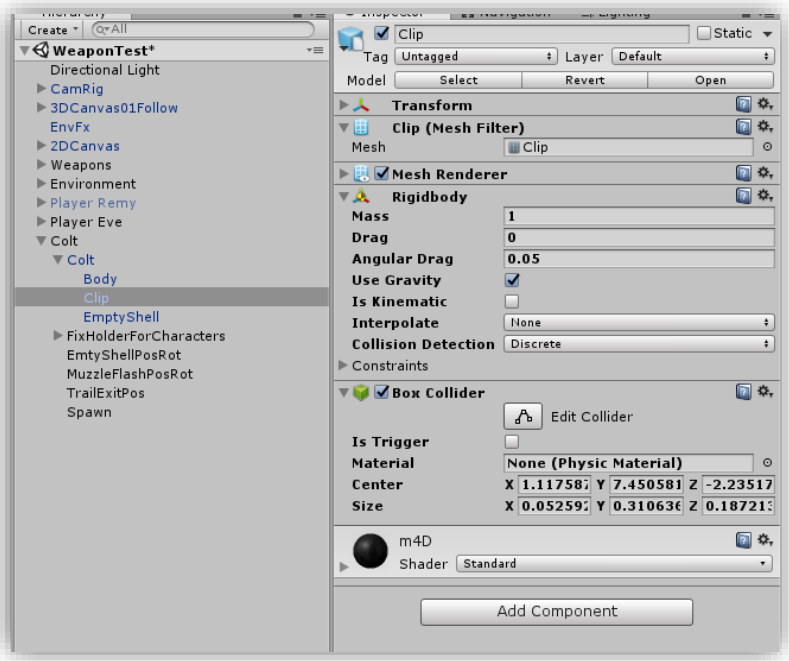
1- I will use a weapon prefab named ***AK47*** for speeding up process. I've dropped prefab named ***AK47*** into the scene, then dropped ***Colt*** model as a child to ***AK47*** prefab. So drop your weapon model into the scene and set it as a child to the prefab you choose, reset position. You can scale it down/up if necessary, or you can change the scale factor of the new weapon model.



2- Rotate the weapon model to match the old model. If you can position the new model to match the weapon handles, it will speed up the fixing holding positions process, if you can't, no problem, all of them can be modified layer. You can delete the old model (which is child of weapon prefab), and old weapon's clip now.

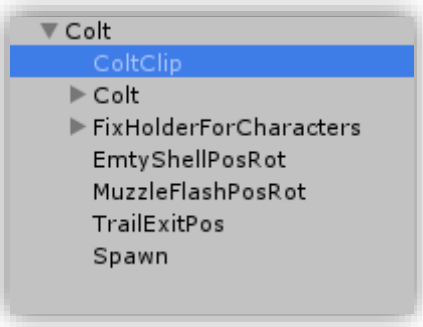
3- Rename the GameObject, modify the box collider if necessary.





4- If the weapon's clip is seperated you will be able to create a rigidbody clip like I will do now. Select the weapon's clip and add these components: ***Rigidbody, Box Collider***. Rename if necessary.

5- Then disable the Clip box collider and set its Rigidbody to Non kinematic.

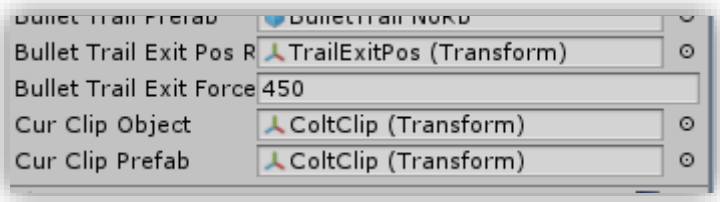


6- Set the clip you see in the hierarchy as child of weapon. Create a Clip prefab by dropping it into the project panel.

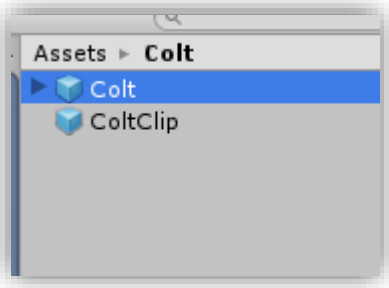
7- Select the weapon and find the fields you see in the right and fill them like shown.

CurClipObject is the child clip object you set in the previous step.

CurClipPrefab is the prefab you have created.



8- Drop the weapon to project panel to create a weapon prefap of the new weapon.

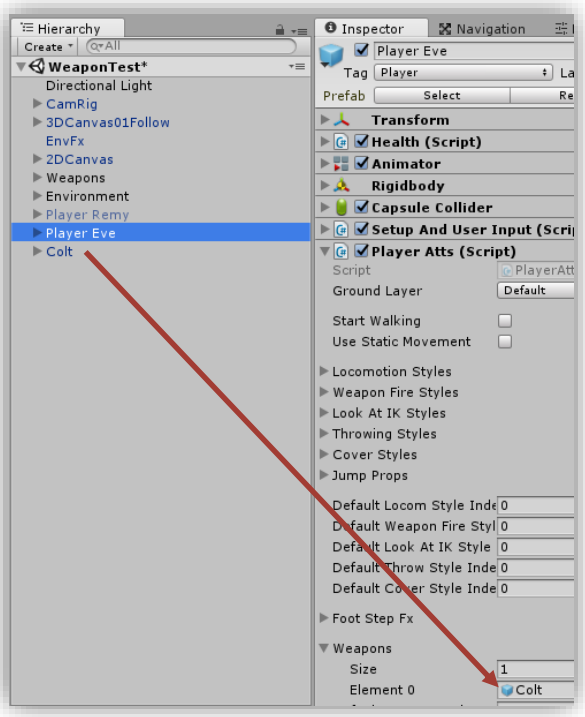


9- Take a look at ***GunAtt*** component of the weapon in the hierarchy, here you can modify every aspect of weapon behaviour. For now, I will only modify basic features like weapon name, clip capacity, gunstyle etc. We can fine tune it in play mode. Select a bullet prefab by assigning a new bullet prefab to ***CurrentProjectilePrefab*** field. (To create a new bullet type refer to defining new bullet section).

10- You can see that a weapon has some child gameobjects. They are used by GunAtt to hold positions. EmptyShellPosRot is used for holding the Instantiate position/rotation of empty weapon shells, TrailExitPos is used to hold trail exit position/rotation, Spawn holds the position of raycasts or Instantite position of projectiles like rocket. So position them to match your new model. Don't modify FixHolderforCharacters yet.

Tip: A weapon can fire specific caliber of bullets. This means if player's weapon clip is empty, ***Weapon System*** will try to search with the same caliber bullet in the ***PlayerAtts ammoBag*** field. Caliber number is used to distinguish the bullets that different weapons using.

Tip: If a weapon is not using raycast projectiles, some of the ***GunAtt*** fields will not be used, which means they will be overriden by projectile like ***Rocket*** projectile of ***RPG***...



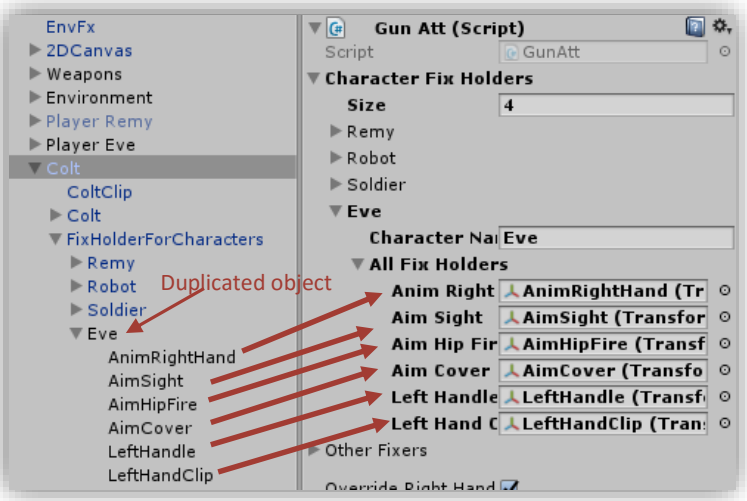
11- I will now assign new weapon to my player to fine tune it.

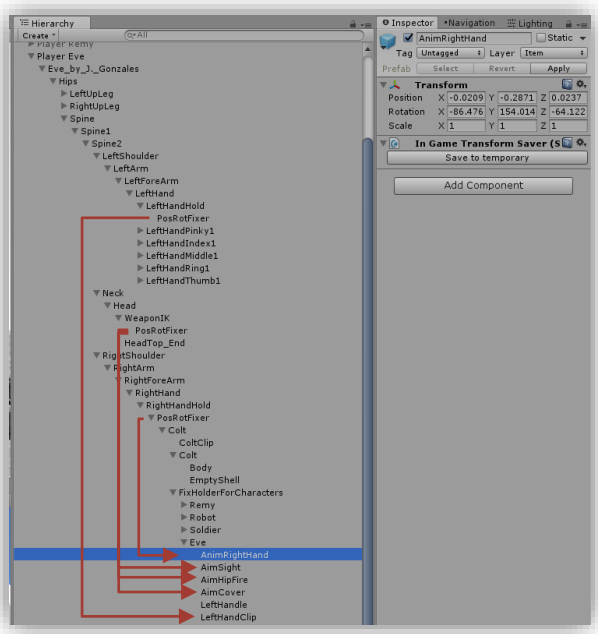
If I hit the play button now, you can see in the picture, my character does not really know how to hold this weapon properly. Now we will fix this quickly.

12- (Skip to 13 If you have not imported new character yet) Stop the play mode now. Select the weapon in the hierarchy and find CharacterFixHolders field of GunAtt from the inspector.

We will need 6 position/rotation hold gameobjects per character. As you can see I've already set up the old weapon to be used by 3 characters. Let's say I want the new weapon to be used by one more character which is Player Eve.

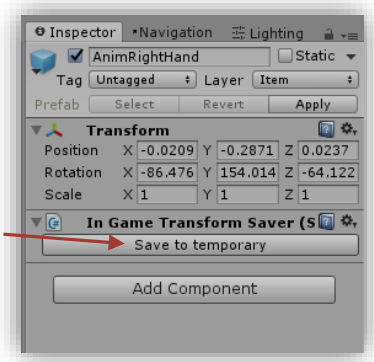
I will increase size of CharacterFixHolders list and I will rename it to Eve. Then I will duplicate one of the already made weapon position holders as shown in the right, lastly I will drop the duplicated gameobjects to the new list item.





13- Start the play mode, pull out your weapon.

First let's fix the weapon in right hand to match the animation. You will need to select the object named PosRotFixer which is child of RightHandHold. Then position it or rotate it from the scene to match it to the player's hand animation. When you finished positioning, copy the Transform component of the PosRotFixer, paste it to AnimRightHand object's transform component like shown in the right.

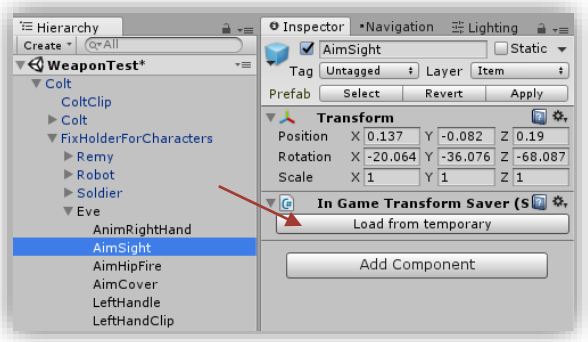


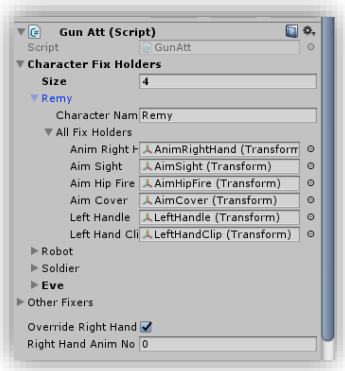
After that hit the Save to temporary button (on the AnimRightHand) so that we will not lose those position/rotation numbers when we stop the play mode.

Fix the other positions : AimSight, AimHipFire, AimCover, LeftHandClip by using PosRotFixers like shown above. Keep in mind that only LeftHandle will be modified directly using itself.

Tip: You can use the player's PressFire2Button when fixing aim positions. (You don't have to press Fire1Button to fix AimHipFirePosition) and Fix cover position while cover aiming.

When you have done finding positions and saved them to temporary, stop play mode then load them from temporary.





Tip: When a character (player or shooter Ai) pulls out the weapon, positioning will be done by the name you entered the characterName field. For example when the player named Player Remy is pulling out weapon or aiming with it, he will use the fields with the matching character name in the GunAtt (Because Player Remy 'Parent' GameObject name contains : Remy)

b. Throwing System

This system is capable of throwing, prefabs that are Instantiated in game. There is a Frag Grenade example in the project.

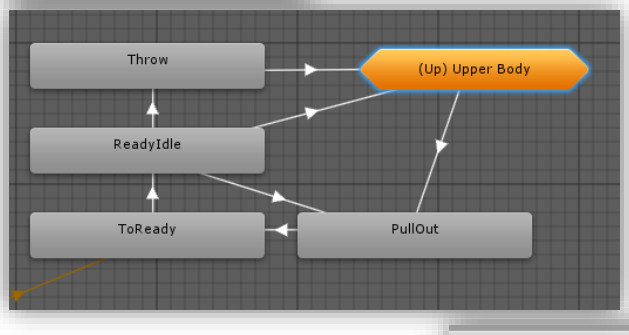
In order for player to throw a prefab, prefab needs to have have 3 main components. Rigidbody, Collider, Exploder. You can modify Exploder component of a throwable to define different throwable behaviours. Rigidbody and Collider somehow optional, but Throw system will need Exploder component of a prefab to work.

b.1 Defining New Player Animations for Different Throwables

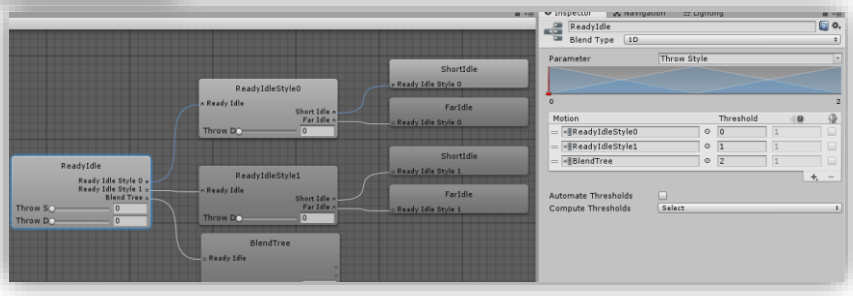
You can also define different player animations for different throwables.

As an example to this I have added one more throw animations that is not used currently.

Throwing is done in the fourth layer of Player's animator.



If you want to add a different player animation for a throwable you will need to follow a similar way the one explained in the weapon section.



Check the picture in the right side. I have added a blend tree to ReadyIdle blend tree to import a player animation. If you dont want to use the Far/Short throw just create a motion field instead. I have set the Threshold number to 2, so that whenever player pulls out this throwable he will use those animations. Modifying other blend trees like PullOut/ToReady is similar to this example.

c. Camera System

This works similar to some of the player systems like Look At system.Generally, this is used by other systems like Weapon, Cover etc.

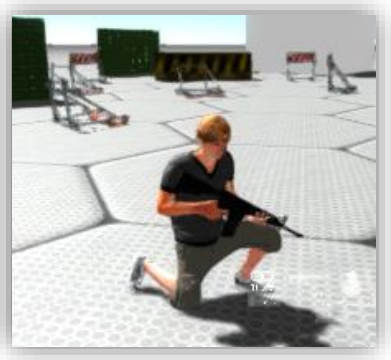
For example Weapon system wants camera to focus to a position, it will tell this intention to Camera, if the camera don't have a more important job, it will do as told.

Currently, this system is completely overridable to Third Person, First Person, Focusing by other systems.

d. Locomotion System

Locomotion system, as the name explains is responsible for movement. This system has 4 states that is overridable by other systems.

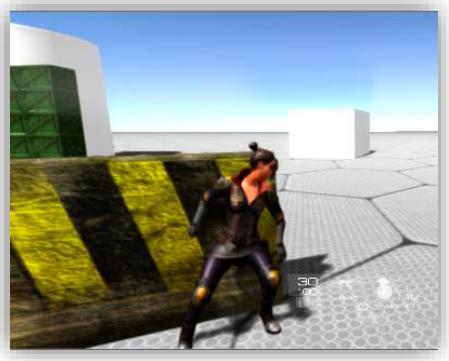
1. Free movement with keys
2. Static movement with keys (Always turned to some transform)
3. Moving with navmeshagent
4. Deactivated state



e. Cover System

This system is responsible from every aspect of player cover.

Cover detection is done by CoverChecker which is child of player. To modify or fine tune cover detection, use CoverTargetLogic component of this gameobject. Other covering parameters can be modified by PlayerAtts component of player as usual.



f. LookAt System

Look At system is defined using Unity's IK system. (Unity's IK system is also used fo weapon arm positioning).

However, by doing some calculations before Unity IK takes place, system is capable of smoother and more flexible humanoid looking while needed.

5. Algorithm and Coding Section

a. Player System Basics

Player systems are theoretically works completely independent from each other. But in the real or game world, generally, this is not possible. So, a system will always try to communicate with other systems.

I will try to explain the workflow of player systems with a couple of examples, hoping that it will help you define your systems later.

Imagine that weapon is pulled out by the Weapon System, player hits the aim button then, again weapon system is playing the aiming animation, but player also wants to move back while aiming so he/she presses the move back button which is for keyboard 'S' key. If both systems work without communicating with each other player will probably aim to the camera itself because Locomotion System is responding the move back button by turning the player. Then you will get some funny behaviours.

Now let's see how IntenseTPS works with cases like these. In our example, when Weapon System starts aiming with a button event, Locomotion system will basically get a message from Weapon System to turn to an object which is always in front of player and camera. After that he will always aim and turn to the position where camera is looking. After that when aiming is finished Weapon System will tell the Locomotion system that it no longer needs to turn to a position. Then locomotion system will return to its default behaviour.

Sounds easy enough ? Let's take a look at another case:

Let's assume player is near a wall and wants to cover, he/she enters the cover by pressing the cover button. Cover system will enter the cover like it should and it will ask the camera to override itself to third person camera state with 'cover specific parameters'. Let's say it has no other request from any other systems and set itself as asked from Cover System. Then player pulls out a weapon and wants to aim, Weapon System will also ask the Camera System to override itself to third person camera but with Weapon System parameters. Which one will the camera choose? In cases like these the answer is the most important one.

The general idea is that a system mustn't ask another system to change its state without an importancy specifier. How will this system use that specifier? Well, that is a problem which system will need to solve by itself.

By following this logic for all the player systems, when you add a new system you don't have to worry too much about if it will work with other systems.

In addition to this, key triggers of systems are not used without an importancy modifier. For example Throwing System starts throwing with a button trigger but sometimes we don't want player to throw a grenade like when entering cover. So the cover system will modify Throwing System's Triggers to disable some or all of its triggers for a short amount of time, so that we will not get glitches or animations mixed badly.

b. Code Samples

If you have read the previous part (Player System Basics), you should know the basic logic behind the player systems. Now let's take a look at some code samples.

I will try to explain how I've integrated some features with code samples. Look at code below.

```
if (userInput.FirstPersonLookDown && !IsFreeLooking && TriggS.LastValue.GetTrigger(LocomotionSystemTriggers.ct_FreeLook))
{
    if (userInput.cameraRig.GetComponent<PlayerCamera>())
    {
        userInput.cameraRig.GetComponent<PlayerCamera>().OverrideCamera(new FirstPersonCameraParams(), -3, c_overrideKey);

        short priority = 1;
        TriggS.Override(c_overrideKey, priority, new LocomotionSystemTriggers(false, false, true, false, false));
        player.SmbWeapon.TriggS.Override(c_overrideKey, priority, new WeaponSystemTriggers(false));
        player.SmbCover.TriggS.Override(c_overrideKey, priority, new CoverSystemTriggers(false));
        player.SmbThrow.TriggS.Override(c_overrideKey, priority, new ThrowSystemTriggers(false));
        player.SmbLookIK.OverrideToDeactivateLookAt(new DeactivatedLookAtParams(player.defaultLookAtIKStyleIndex), priority, c_overrideKey);
        IsFreeLooking = true;
    }
}
else if (IsFreeLooking && (userInput.FirstPersonLookDown || !TriggS.LastValue.GetTrigger(LocomotionSystemTriggers.ct_FreeLook)))
{
    userInput.cameraRig.GetComponent<PlayerCamera>().ReleaseOverride(c_overrideKey);
    TriggS.Release(c_overrideKey);
    player.SmbWeapon.TriggS.Release(c_overrideKey);
    player.SmbCover.TriggS.Release(c_overrideKey);
    player.SmbThrow.TriggS.Release(c_overrideKey);
    player.SmbLookIK.ReleaseOverrideLookAt(c_overrideKey);
    IsFreeLooking = false;
}
```

This code is taken from Locomotion script and its job is to enter to first person camera and exit from it. It can also be written as a part of Camera System, since there were no other button triggered events in the Camera System, it is written as a part of Locomotion System.

First if statement checks to see if first person camera button is pressed then checks if it has already in the first person camera mode and then checks if first person camera trigger is enabled with the `TriggS.LastValue.GetTrigger(LocomotionSystemTriggers.ct_FreeLook)` code... So, if the property named Triggs is not overriden by any other system player will enter first person look mode.

When the first person look mode is activated, the locomotion system will tell the camera to override itself to first person state with a short integer number in this case : -3 which is very small number compared to other events that tells the camera to change its

state. This means that camera will override itself to this state even if it is overriden and changed its state, by another event.

Then this trigger will try to override Weapon System's Triggers which are basically button triggered events like pulling out weapon, aiming etc... Same requests are also sent to Cover System and Throw system, and lastly it will send a request to Look At system to set itself to a deactivated state.

What have we achieved? Well, after entering the first person look mode, the player will not be able to interact with weapon even if he/she presses the related buttons, he/she won't be able enter cover etc...

So if you check the else if statement (conditions to exit first person look mode), player will exit the first person look mode if he/she presses the button or if the `!TriggS.LastValue.GetTrigger(LocomotionSystemTriggers.ct_FreeLook)` condition is true, which means the Locomotion system Triggers property is overriden by another system.