12/21/2020

Imitation Studios

www.imitationstudios.com

Orbital Aiming System

Contact: support@imitationstudios.com

Primary Documentation:

## Video Tutorial Links:

Animator Setup:

https://youtu.be/X7bVl7At9F4

Weapon Setup:

https://youtu.be/b6cYwKunKxM

Character Setup:

https://youtu.be/Irh8IEgjmg0

## Introductions and Announcements:

Hello everyone, before I get into the details of how this documentation is organized, I would like to sincerely thank you for purchasing this asset pack and, therefore, helping support a fellow developer! If you have any questions or concerns, please feel free to email me at support@imitationstudios.com. I enjoy hearing from you, as it gives me a clear direction for the future development of the asset and identifies any documentation clarifications needed or desired!

If you are familiar with some of my previous work, you will know that in past documentations, I listed the variables with their corresponding descriptions in the documentation. This was in addition to complete sets of tooltips. Honestly, this seems incredibly inefficient. So, in this package, I've created a tooltip for every variable but will not relist that information here.

I'm aware that some of you are only reading this because you have encountered the phrase "refer to the documentation". Don't worry. You're in the right place. If a tooltip says, "refer to the documentation", this primarily means it is easier to understand the variable in the context of the relative system rather than individually. These will be explained after the setup sections of this documentation.

Thanks again and please email me if you have any questions or concerns!
support@imitationstudios.com

## Package Overview:

The Orbital Aiming system is comprised of three major parts not including the target. These parts are the character, the orbital exoskeleton, and the weapon.

**The Character:** The character is your own character mesh and components which will need to be set up in the following fashion:

1. The character must contain a modified version of your character controller's animator so that it is compatible with the orbital aiming system. Another animator is included in the package: **"OrbitalExoskeletonAnimator". DO NOT modify this animator** to suit a character controller as it has a separate purpose! More information on setting up the character animator is available in the animator section of this documentation.
2. The character must have the Orbital Exoskeleton Prefab be a child of the character's hip bone.
3. The same gameobject that contains the characters animator should contain the script "Orbital Remote". The "required transforms" section of the Orbital Remote script should be assigned as named. Do not assign parts, like hands or fingers, of the exoskeleton to the orbital remote. These fields are for *your* character. The character settings profile should be assigned as well (See the character setup section for more information). **The Orbital Remote is the primary means of controlling the Orbital Aiming System.**
4. The character must have at least one "Orbital Target Provider" script attached. **DO NOT attach more than one target provider to the Exoskeleton gameobject!** Attaching more than one "Target Provider" to any other gameobject is fine. In fact, it is recommended to attach the target provider to the same gameobject the remote is attached to for easy access. Target Providers will be explained in detail in the Target Provider Section.

**The Exoskeleton:**

**IMPORTANT: If you are experiencing issues with the exoskeleton rapidly changing positions (this will happen if your character has a rigidbody attached to the hip bone, usually this means a ragdoll setup), set the exoskeleton's animator's update mode to "Update Physics". If that does not solve the problem, set the character's hip bone's rigidbody's "Is Kinematic" value to true when in aim mode. You may set the "Is Kinematic" value to false when the character is not in aim mode, or, for a ragdoll setup, right before you enable the ragdoll.**

1. The exoskeleton is prefab located under the prefabs folder of the Orbital Aiming System Package. The exoskeleton should be a child of the character's hip bone.
2. The exoskeleton should have the "OrbitalExoskeletonAnimator" attached.
3. The exoskeleton should have the "Orbital Targeting" script attached. All the "Required Transforms" fields should be assigned using the exoskeleton's armature. The orbital

targeting script controls the angle calculations of the aiming system. This script is controlled by the orbital remote in most circumstances.

4. The exoskeleton should have one, **and only one**, target provider attached to the gameobject that shares the orbital targeting script. **This includes the "Target Relay" script!** Refer to the target provider section for more information.
5. If you are using the universal render pipeline or HDRP refer to the render pipeline section of this documentation.
6. The Exoskeleton may be scaled uniformly if needed.

**The Weapon:**

1. The weapon should be a custom prefab object with the orbital weapon script attached.
2. It is highly recommended that the pivot point of the weapon is the weapon's grip.
3. An empty gameobject should be created, positioned where the left hand should go with its rotation aligned with the weapon (not the hand), and it should be assigned to the "Bind Point" parameter of the orbital weapon script.
4. A weapon profile should be assigned to the orbital weapon script. See the weapon setup section for more information.

## Setting up a new character:

**Note: Modified character profiles should only be used to solve minor differences in characters using the same weapons. If your characters are too physically different, you will need to make two or more weapon profiles for the same weapon and manage their assignation based on the character using them.**

Step 1: Load the "CharacterSetup" scene located under the "UtilityScenes" folder.

Step 2: Place your character in the middle of the scene and disable all its components except for the animator and mesh renderer. This is primarily to prevent the character from moving and therefore includes components related to unity's physics system.

Step 3: Assign the "OrbitalCharacterDemo" animator to the character.

Step 4: Add the orbital remote script to your character and assign your character's armature to the "Required Transforms" fields as named.

Step 5: Right-click in the project window. Click create/Orbital Aiming System/Character Profile.

Step 6: Assign the newly created character profile to the orbital remote script.

Step 7: Add the "Mouse Position Target Provider" to the character and assign the main camera to it.

Step 8: Add the "Orbital Weapon Spawner" script to the character, set weapon prefab to the "OrbitalCharacterSetupWeapon", and set spawn to true.

**Note: If you have scaled the exoskeleton, you must use a different weapon, rather than the default setup weapon, that is the correct scale.**

Step 9: Add the Orbital Exoskeleton prefab to the scene, position and scale it **(UNIFORMLY)** to match your character, and then parent it to the character's hip bone.

Step 10: Set the "Active Target Provider" of the "Orbital Target Relay" attached to the exoskeleton to "Mouse Position Target Provider". Assign the characters "Mouse Position Target Provider" script to the appropriate target provider relay field.

Step 11: Set up your editor window so that you can see both the scene view and game view.

Step 12: Press play and follow the onscreen prompts. If the exoskeleton does not disappear, refer to the render pipeline section of the documentation.

Step 13: Once finished, exit play mode. You may now re-enable all other components.

Step 14: The mouse position target provider may now be switched out as desired. Just do not forget to keep the relay's parameters up to date.

Step 15: You can reassign your own character's animator. If you have not yet adjusted your animator, please see the animator section of this documentation.

Step 16: The orbital weapon spawner may now be removed as desired, see the keeping track of weapons section for more information.

Step 17: The gyro limits and distance correction settings may now be adjusted as desired. See their sections for more information.

**Step 18: There is a gameobject that controls the left forearm twist that is a child of the exoskeleton's left upper arm. The position of this gameobject can be changed as desired.**

Step 19: Create a prefab of the character for use elsewhere and then delete it from the scene.

## Setting up a new weapon:

Step 1: Load the "WeaponSetup" scene located under the "UtilityScenes" folder.

Step 2: Create an empty gameobject with a default transform and name your weapon.

Step 3: Place your custom weapon in the scene so that the grip is centered at the (0,0,0) position. The weapon should ideally point in line with the positive Z-axis of the newly created gameobject, but if you later have problems adjusting the weapon's rotation, try changing the axis if you are able to.

Step 4: Create another empty gameobject and position it so that it is located where the player's left hand should be in relation to the weapon.

Step 5: Parent both the step 4 gameobject and your custom weapon to the step 2 gameobject.

Step 6: Assign the orbital weapon script to the step 2 gameobject.

Step 7: Right-click in the project window. Click create/Orbital Aiming System/Weapon Profile.

Step 8: Assign the newly created weapon profile to the “Weapon profile” field of the orbital weapon script.

Step 9: Assign the step 4 Gameobject to the bind point field of the orbital weapon script.

Step 10: Create a prefab of the weapon and delete it from the scene.

Step 11: Set up your editor window so that you can see both the scene view and game view.

Step 12: If necessary, scale the setup character **(UNIFORMLY)** to match the scale of the character that will be using the weapon.

Step 13: Press play and follow the onscreen prompts.

**Note #1: Finger orientation can be adjusted by modifying the weapon profile directly.**

**Note #2: Be aware of gimbal lock when adjusting rotations.**

Step 14: See the recoil section of the documentation for more information.

## Adjusting Your Character’s Animator:

In order for the Orbital Aiming System to function properly, you will need to modify the animator used by your character controller. A video tutorial is available to explain this process. If you wish to edit your animator, use the following steps:

**Note #1: Reload animations are beyond the scope of this asset pack as they are usually weapon specific. Reload animations can be implemented in a very straightforward manner. Just remember to deactivate aim mode for the reload animation.**

**Note #2: The Orbital Aiming System overrides the animations from the chest up, not the hip bone. While this is desired in most cases as it gives the character more life, thought does need to be given to how the rotation of the hip (mainly side to side) during movement cycles will affect the aiming system, as setting the gyro speed high enough to stay locked on the target during the movement cycle will cause the character to snap between the aiming cycle’s extreme ends. In short, hip bone movement should be minimal in the base animations you intend the character to aim from.**

**IMPORTANT: If you are experiencing issues with the exoskeleton rapidly changing positions (this will happen if your character has a rigidbody attached to the hip bone, usually this means a ragdoll setup), set the exoskeleton’s animator’s update mode to “Update Physics”. If that does not solve the problem, set the character’s hip bone’s rigidbody’s “Is Kinematic” value to true when in aim mode. You may set the “Is Kinematic” value to false when the character is not in aim mode, or, for a ragdoll setup, right before you enable the ragdoll.**

Step 1: Add the following parameters to the animator. **Name them exactly as listed**:

1. Type Float; “OrbitalYGyroRotation”

2. Type Float; “OrbitalXGyroRotation”
3. Type Trigger; “OrbitalFire”
4. Type Float; “OrbitalFireIntensity”
5. Type Float; “OrbitalFireSpeed”
6. Type Bool; “OrbitalAimMode”
7. Type Float; “OrbitalLeftHand”
8. Type Float; “OrbitalLeftHandOpenness”
9. Type Float; “OrbitalRightHand”

Step 2: Add a new layer and name it as desired, but here it will be referred to as “Chest Override Layer”

Step 3: Set the layer’s weight to 1, Mask to the included “UpperBody” asset, blending to “Override”, and IK to true.

Step 4: Create a new default state in the chest override layer. Ensure the motion is listed as “None”. This will be the state in which the aiming system is deactivated and will not override the previous layers.

Step 5: Create a new blend tree in the chest override layer.

Step 6: Set the blend tree’s speed to 0 and set the cycle offset to use the parameter: “OrbitalYGyroRotation”

Step 7: Open the blend tree, disable automate thresholds, and add 8 motion fields.

Step 8: Set the thresholds and animations of the motion fields to the following values:

1. *Threshold*: -67.5; *Animation Clip*: “HumanRig|Aim_Cycle_n67_5”
2. *Threshold*: -45; *Animation Clip*: “HumanRig|Aim_Cycle_n45”
3. *Threshold*: -22.5; *Animation Clip*: “HumanRig|Aim_Cycle_n22_5”
4. *Threshold*: 0; *Animation Clip*: “HumanRig|Aim_Cycle_p0”
5. *Threshold*: 22.5; *Animation Clip*: “HumanRig|Aim_Cycle_p22_5”
6. *Threshold*: 45; *Animation Clip*: “HumanRig|Aim_Cycle_p45”
7. *Threshold*: 67.5; *Animation Clip*: “HumanRig|Aim_Cycle_p67_5”
8. *Threshold*: 90; *Animation Clip*: “HumanRig|Aim_Cycle_p90”

Step 9: Set the blend parameter of the blend tree to use “OrbitalXGyroRotation”

Step 10: Return to the chest override layer and add transitions to and from the blend tree and the empty state.

Step 11: For both transitions set Has Exit Time to false. Add the bool “OrbitalAimMode” as a condition for transitioning into the blend tree (true) and transitioning from the blend tree (false).

Step 12: In the empty state, attach the script “Orbital Aim Mode Deactivator” to it. This script has the single parameter “IK Fade Speed”. This value represents the speed at which the hand IK is deactivated when transitioning from the aiming state.

Step 13: In the blend tree, attach the script “Orbital Aim Mode Activator” to it. This script has two parameters. The IK Fade Speed has the same concept but instead applies to the enabling of hand IK when transitioning into the aim mode. Aim Calc Delay softly limits the rotation for the period of time specified in an effort to prevent the character’s arms from moving through its chest in the animation transition.

**Note: from this point forward setting the animator’s aim mode bool will completely activate and deactivate the aiming system *on a properly set up character with an assigned weapon*. No other functions need to be called by your custom script.**

Step 14: Create a new animator layer. This layer will be referred to as chest additive.

Step 15: Set the layer’s weight to 1, Mask to the included “UpperBody” asset, blending to “Additive”, and IK to true.

Step 16: Create a new default state and ensure the motion field is set to “None”.

Step 17: Create a new blend tree, set its speed to 1, and the speed multiplier to use the parameter “OrbitalFireSpeed”.

**Note: Depending on the complexity of the base character controller, the character’s fire animation and the exoskeleton’s fire animation may come out of sync (the gun will not appear to be moving with the character in the fire animation). Setting the character’s fire blend tree’s base speed value (not the parameter multiplier) to a higher value than 1 may help this issue. If you are experiencing this issue and cannot resolve it, please email support@imitationsudios.com.**

Step 18: Create a transition from the empty state to the blend tree. Set Has Exit Time to false. Set a condition to use the trigger “OrbitalFire”

Step 19: Create a transition from the blend tree to the empty state. Set Has Exit Time to true.

**Note: You may need to adjust the transition’s settings so that it matches the exoskeleton’s fire animation transition located in the “Fire Additive” layer.**

Step 20: Open the blend tree editor, set Automate Thresholds to false, and add 2 new blend tree motion fields. Set their thresholds to be 0.3 and 0.6, respectively. Set the main blend tree parameter to “OrbitalYGryoRotation”

Step 21: Edit the 0.3 threshold blend tree. Set its blend parameter to “OrbitalFireIntensity”. Set Automate threshold to false. Add the following motions and their respective thresholds:

1. *Threshold*: 0; *Animation Clip*: “HumanRig|Fire_135Degrees_Limited”
2. *Threshold*: 1; *Animation Clip*: “HumanRig|Fire_135Degrees”

Step 22: Edit the 0.6 threshold blend tree. Set its blend parameter to “OrbitalFireIntensity”. Set Automate threshold to false. Add the following motions and their respective thresholds:

1. *Threshold*: 0; *Animation Clip*: “HumanRig|Fire_225Degrees_Limited”
2. *Threshold*: 1; *Animation Clip*: “HumanRig|Fire_225Degrees”

Note: To play the fire animation and apply recoil, refer to the public functions section of the documentation. An example of this is given by the “OrbitalTestfire” script.

Step 23: Create 2 new layers: “Left Hand” and “Right Hand”. **In both layers:** Set its weight to 1, IK to true, blending to additive, and mask to left hand and right hand, respectively.

Step 24: **In both layers:** create an empty default state, create a new blend tree, create transitions to and from the empty state and blend tree on the condition “OrbitalAimMode” set to true (blend state) and false (empty state), respectively.

Step 25: **Left-Hand Blend Tree:** Set automate thresholds to false, create three motion fields (listed below), Set the blend parameter to “orbital left hand”, set the entire blend tree’s speed to 0 (not the individual motions), and set the cycle offset parameter to “Orbital Left Hand Openness.”

1. *Threshold*: 0; *Animation Clip*: “HumanRig|Orbital_LH_FingerPosition_Open”
2. *Threshold*: 1; *Animation Clip*: “HumanRig|Orbital_LH_FingerPosition_Grab”
3. *Threshold*: 2; *Animation Clip*: “HumanRig|Orbital_LH_FingerPosition_Grip”

Step 26: **Right-Hand Blend Tree:** Set automate thresholds to false, create two motion fields (listed below), Set the blend parameter to “orbital right hand”.

1. *Threshold*: 0; *Animation Clip*: “HumanRig|Orbital_RH_FingerPosition_Vertical”
2. *Threshold*: 1; *Animation Clip*: “HumanRig|Orbital_RH_FingerPosition_Horizontal”

## Keeping track of the weapon:

**The only included script that will actually spawn a weapon by default is the “Orbital Weapon Spawner”,** but this script is entirely optional. However, if you wish to spawn your own weapon’s you will need to assign them to the orbital remote using the function OrbitalRemote.SetWeapon(OrbitalWeapon orbitalWeapon, out Transform unlatchedGun). The orbital remote may also spawn a weapon through the proper function. See the public functions section for more information.

Once a weapon is spawned, it will exist in one of three states: locked, unlocked, or undefined. **If the weapon is not actively interacting with an orbital remote (if it has not been set), the orbital weapon component should be disabled.** The actual changing of these states is based on the orbital remote’s current hand IK weight value (Note: this is not the actual animator’s IK weight, but the weight desired by the orbital remote). A weapon will be locked if the IK weight is larger than the weapon profile’s “Weapon Gate” parameter. Once locked, it will be unlocked if the IK weight drops below a value of 1. The adjustment speed parameter of the weapon profile changes the speed at which the transform of the weapon transitions between the profile’s values as needed.

Locked Weapon State: In the locked weapon state, the weapon will be parented to the exoskeleton’s hand and its transform will be constantly changed to what is specified by the locked settings of the weapon’s profile.

Unlocked Weapon State: If the weapon is not in the locked state and the weapon profile parameter “Use Unlocked Settings” is enabled, the weapon will be in the unlocked weapon state. In this state, the weapon will be parented to the “Unlatched Weapon Parent” object specified by the orbital remote. This object is usually the character’s hand. The weapon’s transform will be constantly changed to what is specified by the unlocked settings of the weapon’s profile.

Undefined Weapon State: If the weapon is not in the locked state and the weapon profile parameter “Use Unlocked Settings” is disabled, the weapon will be in the undefined weapon state. In this state, the weapon will be parented to the “Unlatched Weapon Parent” object specified by the orbital remote. This object is usually the character’s hand. *The weapon’s transform will not be modified beyond the parenting action of the initial state change.* You may modify the weapon’s transform when it is in this state. The orbital remote’s weapon events are there to help you in this case.

## Target Providers:

For the aiming system to function, the character must have a target provider attached to the exoskeleton. These control where the character will aim, whether it is transform-based, mouse-based, or camera direction based. If you wish to switch target providers during runtime, the relay target provider must be used and the actual target providers must be attached to a separate gameobject, usually the character base parent. Do not attach more than one target provider to the exoskeleton gameobject. If you wish to use target assist, the target’s collider’s gameobjects must have the “orbital target” script attached. The current target providers are:

1. Camera Direction Target Provider: This target provider will cause the character to aim at the intersection of a raycast from a predefined object. Despite the name, this object does not actually have to be a camera. This provider supports target assist and the raycast is fully adjustable through the script’s parameters.

2. Mouse Position Target Provider: This target provider will cause the character to aim at the intersection of a raycast from the mouse position of a predefined camera. This provider supports target assist, usable screen cropping, and the raycast is fully adjustable through the script’s parameters.

3. Transform Target Provider: The most basic of the providers, this will cause the player to aim at a specific transform with an optional global and or relative offset.

4. Orbital Transform Target Provider: Like the transform target provider, this will cause the player to aim at a specified object with the orbital target script attached. This provider’s advantage over the transform target provider is that the offsets are not global relative to the player but are instead based on the individual target’s settings.

5. Orbital Target Relay: When this script is attached to the exoskeleton, the character will use the active target provider linked to it. This script also allows the switching of target providers during runtime via a public function. See the public functions section for more information.

## Gun Firing Animations and Recoil:

The gun's fire animation can be played through the use of the following function (an example is given in the test-fire script):

OrbitalRemote.PlayFireAnimation(bool applyRecoil);

The recoil controls are located in the weapon profile. There are two primary ideas to keep in mind when adjusting the recoil settings: individual and additive recoil. While I will describe these concepts, the best way to understand them is to experiment!

Individual Recoil: Individual recoil is applied to the character with every weapon shot. The recoil vector is defined by the randomized directional weight specified and its magnitude is capped by both the minimum and maximum shot magnitudes. The backward movement of the gun is a normal additive animation with adjustments for its intensity (kick) and speed. The entry speed parameter should be used to synthesize the backward movement of the gun with the directional movement of the gun.

Additive Recoil: Additive recoil is the sum of the recoil of several shots in rapid succession. This is controlled primarily by both the recovery speed parameter and the rate at which you call the fire animation in your custom script. If the character has not fully recovered from one shot before the next is fired, the residual recoil and the recoil from the latest shot will be added together. The additive recoil is limited by the soft cap and hard cap. The hard cap will prevent the recoil from aiming the gun more than the degrees specified. The soft cap will apply the defined multiplier to degree values above it, slowing the rate at which recoil is accumulated.

## Distance Correction:

Because the gun is located by the character's right shoulder and not in the center of its chest, the base aiming animation will set the gun angled aiming slightly into a target focal point located a specific distance away from the player. To improve the accuracy of the gun when aiming at targets, in front of the character, either before or beyond this focal point, the distance correction parameters located on the exoskeleton's orbital targeting script were added. Please note: the orbital aiming system is not always 100% accurate at all angles but is accurate enough that a standard degree of bullet deviation would be unnoticeable. Distance Correction should only be enabled if your character will only be aiming in front of themselves.

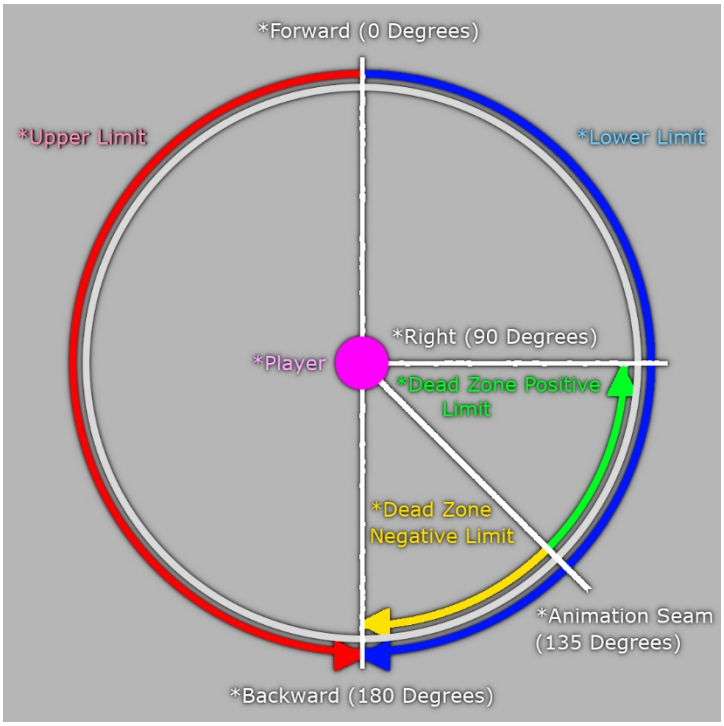
There is a setting on the weapon's profile that will adjust how the character will adjust for projectile drop over distances. These systems are independent of each other.

## Render Pipelines:

If you are using either the universal render pipeline or HDRP you will need to change the material of the exoskeleton via the mesh settings of the orbital targeting script located on the exoskeleton object. **Note: simply changing the exoskeleton's material via its mesh renderer is not adequate as it is changed in a start function.** The appropriate material may be imported into the material folder of the package.

## Aiming Angle Limits (Gyro Limits):

While the degree of the available aiming angle will ultimately be determined by the quality of the character's armature in most scenarios, the orbital aiming system supports 89.9 degrees of upward aiming (from center), 67.5 degrees of downward aiming, and 359.8 degrees of horizontal aiming with the animation seam located at -135 degrees from center. If your character's armature cannot support these angles, the gyro can be limited by the orbital targeting script's parameters located on the exoskeleton. Refer to the image for a graphical representation of the limits. The gyro's speed may also be adjusted.




**Note**: Setting the upper limit to a value less than 180 will void the negative dead zone by capping the lowest "lower limit" to a value of -135.

**Snap Reduction:** In an effort to prevent the character from rapidly snapping from extreme ends of the animation cycle when the target is near the animation seam, the snap reduction parameters have been created. Their function is actually quite simple. When the target is within the angle limit surrounding the animation seam, defined by the snap reduction angle limit parameter, the target's location will be averaged x amount of times as defined by the snap reduction intensity. A higher snap

reduction intensity will reduce snapping but will also make the character slower to react to target movement when within the snap reduction angle window.

## Public Functions:

### Void OrbitalRemote.AttemptChangeTargetProvider(OrbitalTargetRelay.UsedProvider newProvider)

This function will attempt to change the active target provider of the orbital relay script attached to the exoskeleton. It will fail if the attempt to find the target provider relay fails. There is no safeguard in this function against setting a null target provider as the active provider.

**Example:**

*GetComponent<OrbitalRemote>().AttemptChangeTargetProvider(OrbitalTargetRelay.UsedProvider.transformTargetProvider);*

### Void OrbitalRemote.SetCharacterSettings(OrbitalCharacterProfile newSettings)

This function will change the active character profile of the Orbital Remote to the one specified.

**Example:**

*public OrbitalCharacterProfile myProfile;*

*GetComponent<OrbitalRemote>().SetCharacterSettings(myProfile);*

### Void OrbitalRemote.SetWeapon(OrbitalWeapon Weapon, out Transform unlatchedGun)

This function will assign a specified orbital weapon that already exists in the scene to the orbital remote. If the orbital remote already has a weapon assigned to it, its transform component will be outputted.

**Example:**

*public OrbitalWeapon myWeapon;*

*transform weaponOutput;*

*GetComponent<OrbitalRemote>().SetWeapon(myWeapon, out weaponOutput);*

### Void OrbitalRemote.SpawnWeaponFromPrefab(Gameobject prefab, bool destroyReplacedGun = true)

This function will spawn an orbital weapon from a prefab and assign it to the orbital remote. If the orbital remote already has a weapon assigned, it will be destroyed if destroyReplacedGun is not set to false.

**Example 1:**

*public GameObject myWeaponPrefab;*

*GetComponent<OrbitalRemote>().SpawnWeaponFromPrefab(myWeaponPrefab)*

**Example 2:**

*public GameObject myWeaponPrefab;*

*GetComponent<OrbitalRemote>().SpawnWeaponFromPrefab(myWeaponPrefab, false)*

## Void OrbitalRemote.SpawnWeaponFromPrefab(Gameobject prefab, out Gameobject spawnedGameobject, bool destroyReplacedGun = true)

This function will spawn an orbital weapon from a prefab and assign it to the orbital remote. If the orbital remote already has a weapon assigned, it will be destroyed if destroyReplacedGun is not set to false. See the function listed above. This function will also output the spawned gameobject.

**Example:**

*public GameObject myWeaponPrefab;*

*public GameObject myWeaponInScene;*

*GetComponent<OrbitalRemote>().SpawnWeaponFromPrefab(myWeaponPrefab, myWeaponInScene)*

## Void OrbitalRemote.PlayFireAnimation(bool applyRecoil)

This function will set the "OrbitalFire" animator trigger for both the exoskeleton's and the character's animators. Setting applyRecoil to true will also cause the orbital targeting script to apply directional recoil.

**Example:**

*GetComponent<OrbitalRemote>().PlayFireAnimation(true);*

## OrbitalWeapon OrbitalRemote.GetWeapon()

This function will return the orbital weapon assigned to the orbital remote.

**Example:**

*OrbitalWeapon currentWeapon = GetComponent<OrbitalRemote>().GetWeapon();*