

Thank you for your purchase! The most recent documentation can be found [online](#). If you have any questions feel free to post on the [forums](#) or email support@opsive.com.

Introduction

The Third Person Controller is the ultimate third person, networking, and mobile framework. The Third Person Controller is extremely flexible and can be used with Behavior Designer for lifelike AI agents.

The base directory has the following structure:

- Third Person Controller/Demos
- Third Person Controller/Editor
- Third Person Controller/Integrations
- Third Person Controller/Resources
- Third Person Controller/Scripts

The Demos folder contains the various genre demos. The Demos folder does not contain any of the Third Person Controller folder and can safely be removed if you aren't using any of the assets within that folder. If you delete the Demos folder ensure you have replaced all of the animations within the Animator Controller to your own animations.

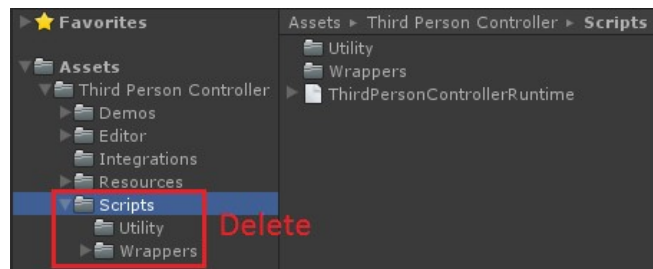
The full source code can be downloaded from [this page](#). Before you import the source code ensure that you have first read [this page](#). In addition, you'll notice that the Third Person Controller includes [wrapper components](#) which allow you to easily switch between the assembly and source code version without losing any references.

Importing Source Code

When the Third Person Controller is downloaded from the Asset Store or Opsive it uses an assembly instead of the source code. If you want to download the source code you can download it from [this page](#). We hate to have to package the Third Person Controller this way, but this is done to reduce piracy. Before you import it you must make sure you delete the scripts folder located at

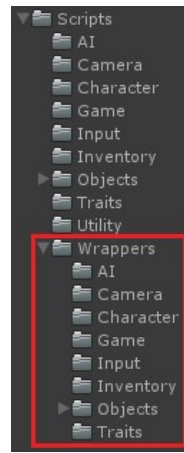
/Third Person Controller/Scripts

If you do not delete this folder and import the source code package you will receive compiler errors.



Wrappers

As you are working with the source code you'll notice that there are two sets of components with the same name. This is done to allow you to easily move between the assembly and the source code version without Unity losing any component references. When you add a new Third Person Controller component make sure you are using the "Wrapper" version of the component. These components can be found in the /Third Person Controller/Scripts /Wrappers folder and use the namespace Opsive.ThirdPersonController.Wrappers. If you do not use these components there is a chance that Unity will lose references to the component within your scene or prefabs if you switch to the source code version.



Camera

The CameraController component controls the camera's position and rotation for a third person or top down view. The ThirdPersonView parameter will specify if the camera is in third person mode or top down. When the camera is in third person mode, the camera will smoothly follow the character as the character moves. If the the [RigidbodyCharacterController](#) is in combat mode, the camera will always be behind the player. When the controller is in adventure mode the camera will be able to freely rotate around the character. If the RigidbodyCharacterController is not set to AlwaysAim, the CameraController will make the character aim when the camera zooms in. Top down mode will follow the character from a birds eye view.

The RigidbodyCharacterController uses FixedUpdate to allow smooth movement with physics. Because of this, the CameraController can't do a full update within LateUpdate otherwise the character will appear as if they are jittering when moving. To circumvent this the CameraController updates in both LateUpdate and FixedUpdate. The CameraController will change positions within LateUpdate if the MoveSmoothing value is zero. The CameraController will change positions within FixedUpdate when the MoveSmoothing value is greater than zero. LateUpdate always updates the input and FixedUpdate always rotates the camera. This method completely prevents the character from jittering.

When the character dies the CameraController will receive the OnDeath event. The CameraController will then either go into a death orbit mode or in a LookAt mode. The death orbit will smoothly circle around the character's position. If the death orbit mode is disabled then the CameraController will just continue to look at the character and not change positions.

The CameraController component works with the CameraMonitor component in order to communicate to other components about the camera's current state. For example, CharacterIK uses to a [SharedMethod](#) to communicate with the camera to know which direction to aim. If you want to use your own custom camera you can assign the CameraMonitor component and everything else will continue to work.

Third Person View

Is the camera using a third person view? Use top down if false

Update Mode

Determines when the camera will update its movement

Init Character On Start

Should the character be initialized on start?

Character

The character that the camera is following

Anchor

The transform of the object to look at

Auto Anchor

Should the anchor be assigned automatically based on the bone?

Auto Anchor Bone

The bone in which the anchor will be assigned to if automatically assigned

Crosshairs Location

The normalized crosshairs location on the screen

Min Pitch Limit

The minimum pitch angle (in degrees)

Max Pitch Limit

The maximum pitch angle (in degrees)

Min Yaw Limit

The minimum yaw angle while in cover (in degrees)

Max Yaw Limit

The maximum yaw angle while in cover (in degrees)

Move Smoothing

The amount of smoothing to apply to the movement. Can be zero

Turn Smoothing

The amount of smoothing to apply to the pitch and yaw. Can be zero

Turn Speed

The speed at which the camera turns

Can Turn In Air

Can the camera turn while the character is in the air?

Camera Offset

The offset between the anchor and the location of the camera

Normal FOV

The normal field of view

Allow Zoom

Can the camera zoom?

Zoom Turn Smoothing

The amount of smoothing to apply to the pitch and yaw when zoomed. Can be zero

Zoom Camera Offset

The offset between the anchor and the location of the zoomed in camera

Zoom FOV

The zoomed field of view

FOV Speed

The speed at which the FOV transitions between normal and zoom

Scope Turn Smoothing

The amount of smoothing to apply to the pitch and yaw when in scope. Can be zero

Scope Camera Offset

The offset between the anchor and the location of the scoped camera

Scope FOV

The scoped field of view

Disable Renderer Distance

Disable the character's renderer when the camera gets too close to the character. This will prevent the camera from clipping with the character

Collision Radius

The radius of the camera's collision sphere to prevent it from clipping with other objects

Recoil Spring

The speed at which the recoil increases when the weapon is initially fired

Recoil Dampening

The speed at which the recoil decreases after the recoil has hit its peak and is settling back to its original value

Death Anchor

Optionally specify an anchor point to look at when the character dies. If no anchor is specified the character's position will be used

Use Death Orbit

When the character dies should the camera start rotating around the character? If false the camera will just look at the player

Death Orbit Rotation Speed

The speed at which the camera rotates when the character dies. Used by both the death orbit and regular look at

Death Orbit Move Speed

The speed at which the death orbit moves

Death Orbit Distance

How far away the camera should be orbiting the character when the character dies

Handlers

The Third Person Controller framework contains many components which have the word “Handler” in their name. These classes are designed to take input from the keyboard and pass it on to their respective controller class. As an example, the CameraController component uses the CameraHandler component to get the pitch and yaw input for the camera. The handler components do not know that the controller component exists. The framework is designed this way to easily allow it to be used for AI. The AI isn’t controlled by player input so it does not need the corresponding handler class.

AI Integration

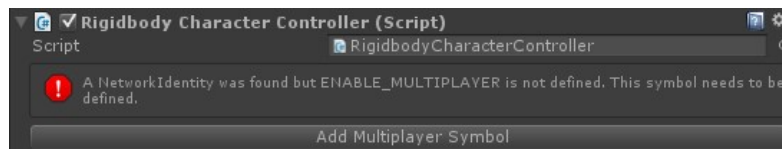
The Third Person Controller is integrated with [Behavior Designer](#) to allow your character to be controlled by an AI within behavior trees. For those who are new to behavior trees, AAA games use behavior trees to create a lifelike AI. The full integration task list is available on the Behavior Designer [documentation page](#). A description of the sample behavior tree created for this integration is listed on [this page](#).

If your character is going to be used for AI then it does not need to respond to human player input. Therefore the PlayerInput component does not need to be added to the character. The [Character Builder](#) will not add the PlayerInput component if the Is AI Agent option is enabled.

Networking

The Third Person Controller supports Unity's networking solution introduced with Unity 5.1. Networking is a complex topic so it is highly recommended that you first go through Unity's [networking documentation](#) before continuing. The Third Person Controller network integration uses an authoritative server on Unity's HLAPI layer. Unity 5.1.2 or later is required for the networking integration to work.

There are two extra steps to make your character a networked character. The first step is to add the NetworkIdentity, NetworkTransform, and NetworkAnimator components to your character. The [Character Builder](#) will automatically add these components if Is Networked Character is enabled within the first step of the wizard. The second step is to add the ENABLE_MULTIPLEPLAYER [compiler definition](#) to your PlayerSettings. This definition can be added manually or automatically added by clicking the "Add Multiplayer Symbol" button within the RigidbodyCharacterController component:



The Third Person Controller includes a demo scene to help get you started with networking. This demo scene can be setup by:

1. Import the Third Person Controller into a Unity 5.1.2+ project
2. Import the [source code](#)
3. Open the Third Person Controller/Demos/Network scene
4. Add the ENABLE_MULTIPLEPLAYER symbol (see above)
5. Play

Object Builders

The disadvantage of having very modular components is that there are a lot of components and it can take some work to initially setup your objects. To help alleviate this problem the Third Person Controller includes the [Character Builder](#), [Item Type Builder](#), and [Item Builder](#). These builders will walk you through the most commonly changed options and add all of the components for you. These values can be changed at any point in time after you have created the object.

Character

The Character Builder is an editor script which will add all of the character-related components to your new character. The Character Builder can be accessed from the Start Window or the Tools menu. The Character Builder consists of two steps in order to create your character.

The screenshots below use the Ethan model available from the [Standard Assets package](#).

Step 1, Base

Base (1/2)

This wizard will assist in the creation of a new character. The values used within this wizard can be changed after the character has been created.

Character:

Is AI Agent ☐

! Is this character going to be used for AI? Some components (such as PlayerInput) do not need to be added if the character is an AI agent.

Is Networked ☐

! Is this character going to be used on the network with Unity 5's multiplayer implementation?

Previous Next

Character

The GameObject that will be used at the character. This is the base layer of your character and will have the majority of the components added to it.

Is AI Agent

Is this character going to be used for AI? Some components (such as PlayerInput) do not need to be added if the character is an AI agent.

Is Networked

Is this character going to be used on the network with Unity 5's networking implementation?

Step 2, Components

Components (2/2)

Specify the type of movement and add any optional components.

Movement Type:

! Combat movement allows the character to move backwards and strafe. If the character has a camera following it then the character will always be facing in the same direction as the camera. Adventure movement always moves the character in the direction they are facing and the camera can be facing any direction. Top down movement moves rotates the character in the direction of the mouse and moves relative to the camera.

Add IK ☒

Add Ragdoll ☒

! Unity's Ragdoll Builder will open when this wizard is complete.

Previous Build

Movement Type

Combat movement allows the character to move backwards and strafe. If the character has a camera following it then the character will always be facing in the same direction as the camera. Adventure movement always moves the character in the direction they are facing and the camera can be facing any direction. Top down movement moves rotates the character in the direction of the mouse and moves relative to the camera. More information is available on the [controller page](#).

Add IK

Should the CharacterIK component be added?

Add Ragdoll

Do you want to add a ragdoll to the character?

If Add Ragdoll is enabled, as soon as Build is clicked the Character Builder will open Unity's Ragdoll Builder. All of the character's transforms will automatically be added to the Ragdoll Builder fields:

Make sure your character is in T-Stand.
Make sure the blue axis faces in the same direction the chracter is looking.
Use flipForward to flip the direction

Root	char_ethan_Hips (Transform)	○
Left Hips	char_ethan_LeftUpLeg (Transform)	○
Left Knee	char_ethan_LeftLeg (Transform)	○
Left Foot	char_ethan_LeftFoot (Transform)	○
Right Hips	char_ethan_RightUpLeg (Transform)	○
Right Knee	char_ethan_RightLeg (Transform)	○
Right Foot	char_ethan_RightFoot (Transform)	○
Left Arm	char_ethan_LeftArm (Transform)	○
Left Elbow	char_ethan_LeftForeArm (Transform)	○
Right Arm	char_ethan_RightArm (Transform)	○
Right Elbow	char_ethan_RightForeArm (Transform)	○
Middle Spine	char_ethan_Spine1 (Transform)	○
Head	char_ethan_Head (Transform)	○
Total Mass	20	
Strength	0	
Flip Forward	<input type="checkbox"/>	

Create

Your character is now created and is ready to be used!

Item Type

An [ItemType](#) is used by the Inventory and Animator to identify a particular Item. The Item Type Builder will create this ItemType for you. The Item Type Builder can create any of the three ItemTypes: [Primary](#), [Consumable](#), and [Secondary](#). The PrimaryItemType is shown below:

This builder will guide you through the ItemType creation process. An ItemType is used by the Inventory and is used to map a particular Item to the Inventory.

Type Primary

Primary

A Primary ItemType is any item that can be equipped by the character.

Consumable Item PistolBullets

Capacity 18

Build

Item

An [Item](#) is anything that can be used by the character. The Item Builder will add the required components and can add the Item to the character automatically.

This builder will create a new Item of the specified type.

Item Type Pistol

Base Pistol

Item Name Pistol

Assign To Doug

Hand Right

Add to Default Loadout ☒

Type Shootable

Assign

Item Type

The ItemType asset used to identify the item for the Inventory and Animator

Base

The base GameObject of the Item. This GameObject can be empty if creating a ThrowableItem

Name

The Item Name specifies the name of the Animator substate machine. It should not be empty unless you only have one item type

Assign To

The GameObject of the character that the Item should be assigned to

Hand

The hand that the Item should be assigned to

Add to Default Loadout

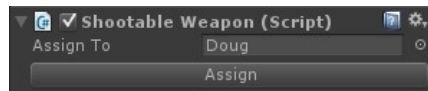
Should the ItemType be added to the character's Default Loadout?

Type

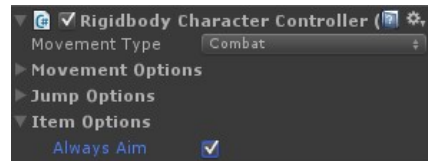
The type of item, a shootable, throwable, or melee

Positioning

After the Item has been created it will need to be correctly positioned in the character's hand. If you assigned the Item to the character with the Item Builder it will already be parented to the dominant hand specified with the Character Builder. If the Item is stored as a prefab within your project you have the option of assigning it to an existing character within the Item's inspector:



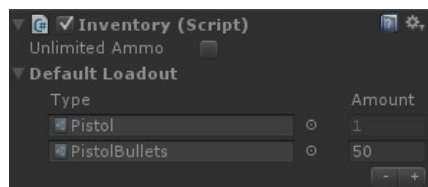
Now the item can be positioned. When positioning the Item the goal is to have the Item aiming at the crosshairs. The first step to accomplish this is to turn on the AlwaysAim property on the RigidbodyCharacterController. This will make the character always extend his arms to aim and is easier to orient the weapon.



In addition to enabling AlwaysAim, the CharacterIK component should be disabled. This will prevent the IK from rotating the limbs while positioning the Item.



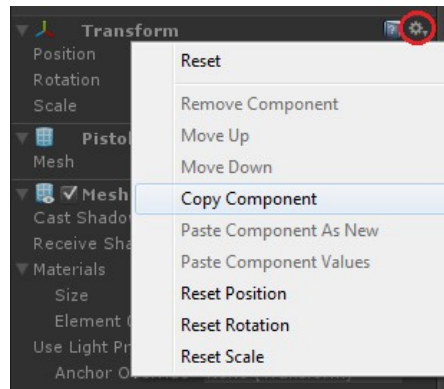
The next step is to assign the Item's ItemType to the Inventory's default loadout so the character will have the Item at the start.



Now that the controller is set to AlwaysAim and the ItemType has been added to the Inventory's Default Loadout the Item is ready to be positioned. Start the game within Unity and find your character and the Item. It should be enabled and the character's arms should be in the aim position. The next step is to manually position the Item so it is facing forward.



Once you are satisfied with the Item's position and rotation go to the Item's Transform component and copy the values (available under the gear icon). This will copy the position and rotation values to the clipboard.



End the game and select the gear icon again. Instead of copying the values select “Paste Component Values”. Your item will now be positioned correctly the next time you start the game.

How To

The following sections describe how to setup a particular feature. Please post on the forum if you'd like to suggest a new How To topic.

[Interactive Object Setup](#)

[Ladder Setup](#)

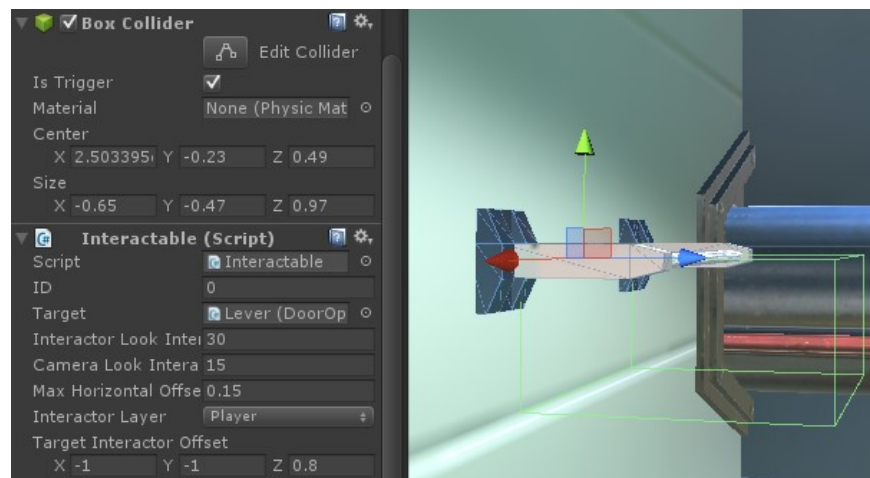
[Replace the Aim Animation](#)

[Replace the Used Animation](#)

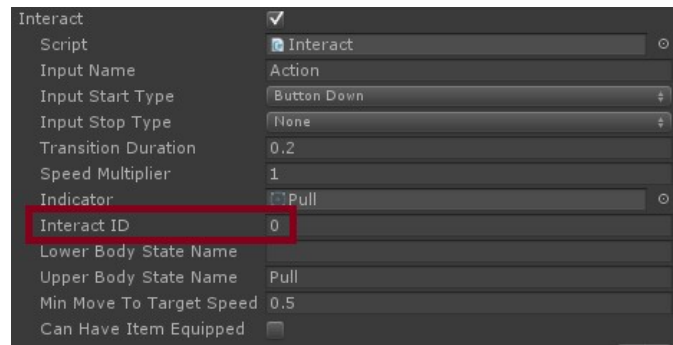
Interactive Object Setup

There are two main components involved in setting up an [interactable object](#). The object that can start the interaction (the Interactable), and the object that does the interaction (the Interactable Target). In the Third Person Shooter demo, the object that starts the interaction is the switch and the object that does the interaction is the moving platform.

In this example we will use the adventure scene to open a door after interacting with another object. In this example the other object is a lever, but it could be anything. The first step is to add the Interactable component to the object that you want to use to start the interaction:



The Interactable component has various settings for determining if the character can interact with the object. It can be any MonoBehaviour as long as it implements to the [Interactable interface](#). In addition, it has a trigger collider which is used to ensure the character is near the Interactable object. The Interactable component has two especially important values: the ID and Target. The ID is used by the character controller to determine which [Interact ability](#) to use when an interaction takes place. Multiple Interact abilities can be placed on the character. This ID maps a particular Interact ability to a particular Interactable object. A value of -1 can be used if there are not going to be multiple types of interactions.



The second import value on the Interactable component is the Target. The Target is the object that does the actual interaction. As soon as all of the settings are satisfied (such as the character looking at the Interactable object and being in the correct layer), this Target object will actually perform the interaction. In our example, the Target object is the actual lever:



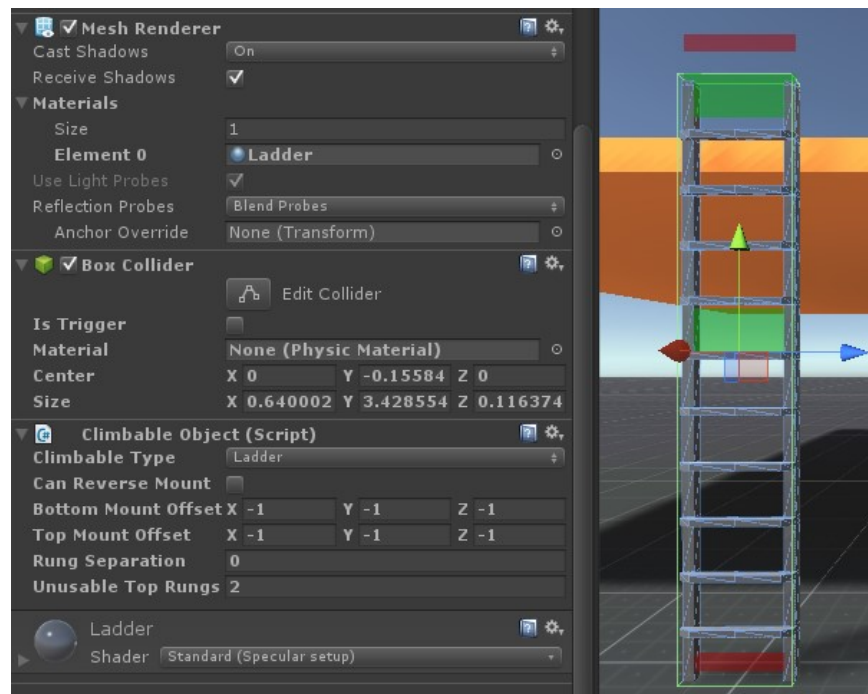
The Interactable Target can be any MonoBehaviour that implements the [InteractableTarget interface](#). The Door Open Interactable component is used in this example. This component will play an animation and toggle the light state when the Interactable component is interacted with.

Ladder Setup

The character is able to climb ladders if the [Climb ability](#) has been added to the Rigidbody Character Controller. In order for the character to climb a ladder, the ladder object must have the [Climbable Object](#) component added. For this example we will start with a basic scene that just has three objects: the floor, upper platform, and ladder.

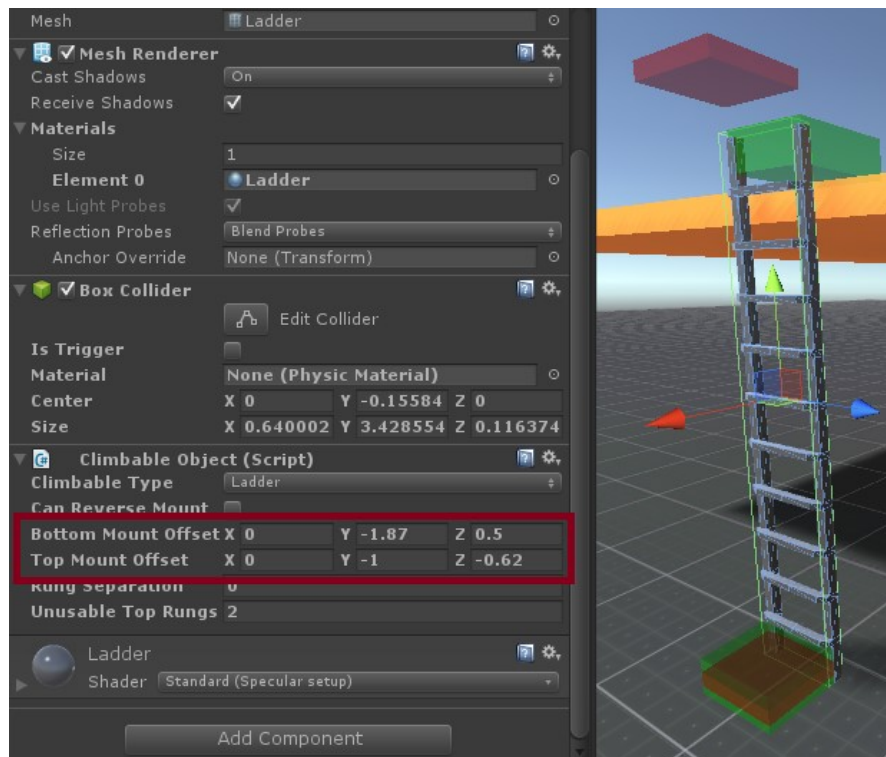


The first step is to add the Climbable Object component to the ladder. This component tells the ability that the object can be climbed and specifies any parameters for the object. When a Climbable Object is selected there are editor gizmos to show the mounting and dismounting positions.

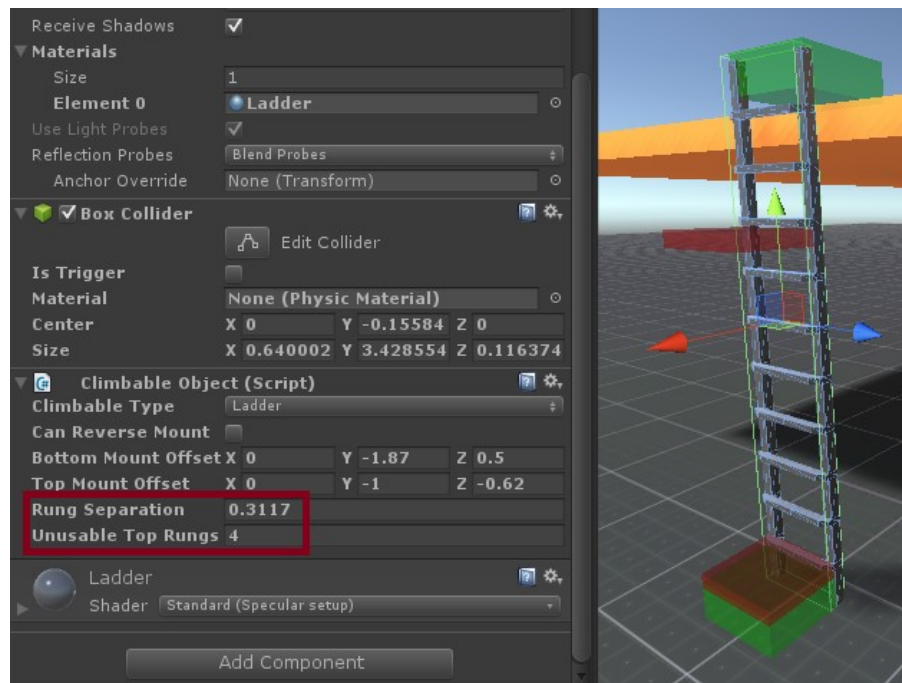


The green gizmos indicate mounting positions and the red gizmos indicate dismounting positions. When the character is on the ladder the blue gizmos will indicate the player position. The actual mounting or dismounting occurs at the center of the vertical space.

The first thing that needs to be modified is the mounting position so there is a green gizmo at the bottom of the ladder and a green gizmo at the top of the ladder. The mount offsets are in local space relative to the ladder Transform. A value of -1 for the mount/dismount offset indicates that the value is not used - the character's position for that axis will be used. The character should be centered on the ladder when mounting so an offset of 0 is specified for the x on both the top and bottom offset. The Bottom Mount Offset is too high so a value of -1.87 is specified to bring the mount position down. The Z offset specifies the distance away from the ladder that the character should start to play the mount animation. A value of 0.5 for the Bottom Mount Offset and a value of -0.62 is used for the Top Mount Offset. After each value change the gizmos will update to reflect their new position:



With the mount offsets set, it is now time to set the dismount position. The ladder is a unique Climbable Object because the dismount offset do not need to be explicitly set. This value can instead be computed based on the Run Separation and the Unusable Top Rungs. The Run Separation is the number of units in between ladder rungs. For this ladder the value is 0.3117. The Unusable Top Rungs specifies the number of rungs that should only be used if the character is dismounting. As soon as the character's position passes the top dismount gizmo the character will start to play the dismount animation. We used a value of 4 for this ladder. The dismount gizmos will update to reflect the changes:

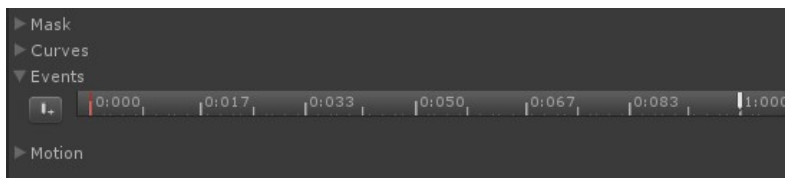


The Climbable Object is now setup to be climbed. The Climb ability still needs to be added to the character in order for the character to actually be able to climb the ladder. No special values need to be setup on this ability to allow the character to climb - just ensure the Climbable Layer specifies the same layer that the Climbable Object is in. Once this is complete your character will be able to mount and dismount on a ladder!



Replace the Aim Animation

Changing an animation is as easy as replacing the clip within the [Animator Controller's](#) state or blend tree. There is one additional step necessary in order to get the aim animation to work. When the character starts to aim, the controller needs to know when the aim animation is done playing so the item can then be used. This notification is sent through [Animation Events](#). If we look at one of the aiming animations, such as the Assault Rifle Aim, we'll see that there is an animation event setup (indicated by the vertical white line near 1:00 on the timeline):



Opening this event we'll see:

Function	ExecuteEventNoUpperTransition
Float	0
Int	0
String	OnAnimatorAiming
Object	None (Object)

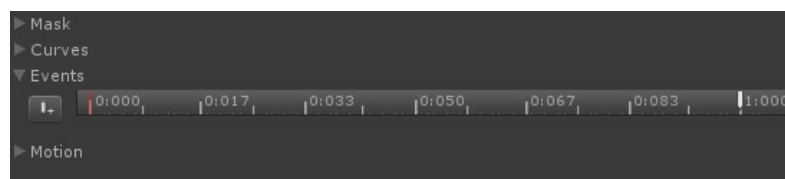
Function: ExecuteEventNoUpperTransition

String: OnAnimatorAiming

These parameters let the controller know that the character is done aiming and is ready to use the item.

Replace the Used Animation

Changing an animation is as easy as replacing the clip within the [Animator Controller's](#) state or blend tree. There is one additional step necessary in order to get the used animation to work. When the character uses an item (such as firing a weapon), the controller needs to know when the actual used event takes place. This prevents the character from using the item too early. This notification is sent through [Animation Events](#). If we look at one of the used animations, such as the Assault Rifle Attack, we'll see that there is an animation event setup (indicated by the vertical white line near 1:00 on the timeline):



Opening this event we'll see:

Function	ItemUsed
Float	0
Int	0
String	
Object	None (Object)

Function: ItemUsed

Int: 0

These parameters let the controller know that the character has used an item. The Int parameter indicates the type of item used. A value of 0 indicates a primary item, 1 indicates a secondary item, and 2 indicates a dual wield item.

Character

The character GameObject is any humanoid character that is controlled by a player or AI. The following components are used by the character:

[Controller](#)

[Abilities](#)

- [Climb](#)
- [Cover](#)
- [Damage Visualization](#)
- [Fall](#)
- [Height Change](#)
- [Ledge Strafe](#)
- [Interact](#)
- [Jump](#)
- [Push](#)
- [Roll](#)
- [Speed Change](#)
- [Swim](#)

- [Vault](#)

[Animator Monitor](#)

- [Animation Events](#)

[IK](#)

[Footsteps](#)

[Health](#)

[Respawner](#)

[Ragdoll](#)

[Spawning](#)

Controller

The RigidbodyCharacterController component controls all of the character's movement. At a high level there are two different types of movement: combat movement and adventure movement. With combat movement the camera is always behind the character's back and this allows the character to strafe and move backwards. With adventure movement the character can move in any direction and allows for a free camera movement. In the adventure case the character never strafes or moves backwards – they will always rotate to face in the direction that they are moving.

The controller uses a Rigidbody for allow for physics-based movement. The controller can optionally use [root motion](#) from the Animator in order to do the actual movement. It will also respond to external forces. An example of an external force includes if there is a nearby explosion and that explosion applies Rigidbody forces to the character.

The RigidbodyCharacterController provides basic movement functionality but the unique functionality comes through the [Ability](#) system. The ability system can give the character any functionality. Some of the included abilities include jumping, taking cover, or performing a roll. These abilities have the option of completely overriding the RigidbodyCharacterController behavior or they can work in addition to. For example, the Cover ability takes complete control of the RigidbodyCharacterController while the Spring ability just modifies a single Animator parameter to make the character move faster.

The character's controller can easily be setup using the [Character Builder](#).

Movement Type

A popup specifying the movement type. Combat movement allows the character to move backwards and strafe. If the character has a camera following it then the character will always be facing in the same direction as the camera. Adventure movement always moves the character in the direction they are facing and the camera can be facing any direction. Top down movement moves rotates the character in the direction of the mouse and moves relative to the camera.

Use Root Motion

Should root motion be used?

Root Motion Speed Multiplier

The multiplier of the root motion movement

Rotation Speed

The speed that the character can rotate

Fast Rotation Speed

The speed that the character can rotate while aiming

Ground Speed

The speed while on the ground and not using root motion

Air Speed

The speed while in the air

Ground Dampening

The amount of dampening force to apply while on the ground

Air Dampening

The amount of dampening force to apply while in the air

Ground Stickiness

Force which keeps the character sticking to the ground while stationary

Skin Width

The additional width of the character's collider

Skin Moving Platforms Stickiness

An extra distance used to determine if the player is on the ground while on a moving platform

Movement Constraint

Optionally restrict the x or z position

Min X Position

If restricting the x axis, the minimum x position the character can occupy

Max X Position

If restricting the x axis, the maximum x position the character can occupy

Min Z Position

If restricting the y axis, the minimum y position the character can occupy

Max Z Position

If restricting the y axis, the maximum y position the character can occupy

Max Step Height

The maximum height that the character can step

Step Speed

The offset relative to the character's position that should be used for checking if a step exists

Step Offset

The vertical speed that the character moves when taking a step

Slope Limit

The maximum slope angle that the character can move on (in degrees)

Always Aim

Should the character always aim?

Item Use Rotation Threshold

The character will rotate to face in the direction of the camera when using an item. If the character is not facing in the correct direction when trying to use an item, they will automatically rotate until an angle less than this value.

Item Forcibly Use Duration

The duration that the character should forcibly use the item

Dual Wield Item Forcibly Use Duration

The duration that the character should forcibly use the dual wielded item

Grounded Friction Material

The friction material to use while on the ground

Step Friction Material

The friction material to use while stepping

Slope Friction Material

The friction material to use while on a slope

Air Friction Material

The friction material to use while in the air

Abilities

The ability system allows the character to add new functionality without touching the `RigidbodyCharacterController` code at all. The abilities can have complete control over the `RigidbodyCharacterController` or can work along the side of the `RigidbodyCharacterController`. For example, the Vault ability completely prevents the controller from updating when it is active. On the other end of the spectrum, the Speed Change (sprint) ability works with the controller and just modifies an `Animator` parameter when the character should be sprinting.

New abilities can be added by deriving your new class from the `Ability` class. This new ability will then automatically show up under the abilities drop down within the `RigidbodyCharacterController` component inspector. Abilities have a priority system in that the higher the ability is on the list, the higher priority it has. For example, lets say that the crouch ability has a higher priority than the jump ability. If this is the case then the jump ability would not be able to play while the crouch ability is active because the crouch ability will override any lower priority abilities.

The following API can be used by the abilities. Methods that return `bool` indicate if the `RigidbodyCharacterController` should stop execution of that method with its own functionality.

```
// Can this ability run at the same time as another ability?
public bool IsConcurrentAbility()
```

```
// Executed on every ability to allow the ability to update.
// The ability may need to update if it needs to do something when inactive or show a GUI icon when the ability can be started.
public void UpdateAbility()

// Can the ability be started?
public bool CanStartAbility()

// Can the inputted ability be started?
public bool CanStartAbility(Ability ability)

// Can the ability be stopped?
public bool CanStopAbility();

// The ability has been started.
public void AbilityStarted()

// Returns the destination state for the given layer.
public string GetDestinationState(int layer)

// Returns the duration of the state transition.
public float GetTransitionDuration()

// Can the ability replay animation states?
public bool CanReplayAnimationStates()

// The ability has stopped running.
public void AbilityStopped()

// Moves the character according to the input.
public bool Move(ref float horizontalMovement, ref float forwardMovement, Quaternion lookRotation)

// Perform checks to determine if the character is on the ground.
public bool CheckGround()

// Apply any external forces not caused by root motion, such as an explosion force.
public void CheckForExternalForces(float xPercent, float zPercent)

// Ensures the current movement is valid.
public bool CheckMovement()

// Apply any movement.
public bool UpdateMovement()

// Update the rotation forces.
public bool UpdateRotation()

// Update the Animator.
public bool UpdateAnimator()

// The Animator has changed positions or rotations.OnAnimatorMove method?
public void AnimatorMove()

// Does the ability have complete control of the Animator states?
public bool HasAnimatorControl(int layer)

// Can the character have an item equipped while the ability is active?
public bool CanHaveItemEquipped()

// The character wants to interact with the item. Return false if there is a reason why the character shouldn't be able to.
public bool CanInteractItem()

// The character wants to use the item. Return false if there is a reason why the character shouldn't be able to.
public bool CanUseItem()

// Should the upper body IK be used?
public bool UseUpperBodyIK()
```

Abilities can be started and stopped by an input button or automatically. For example, the Crouch ability should only be started when the player taps the Crouch button. On the other hand, the Fall ability should be triggered when the character is in the air and has a negative y velocity.

Input Name

The button name that can start or stop the ability

Input Start Type

Specifies how the ability can be started from input

Input Stop Type

Specifies how the ability can be stopped from input

Transition Duration

The length of time it takes to transition to the ability

Speed Multiplier

The Animator multiplier of the state

Indicator

The sprite indicator used to specify that the ability can start or is active

Climb

The Climb ability allows the character to climb up vertical or horizontal objects. This includes ladders, vines, or pipes. The Climb ability works closely with the [ClimbableObject component](#) in order to successfully climb over the object.

Most of the Climb setup is defined on the ClimbableObject rather than the Climb ability. The StartClimbMaxAngle and StartClimbMaxDistance values specify the angle and the distance away from the ClimbableObject that the character can start to climb. The SearchRadius is used to determine how large of a raycast should be performed to find the ClimbableObject, specified by the ClimbableLayer. As soon as the Climb ability is activated the character will move to the mounting position at a minimum normalized speed of Min Move To Target Speed.

Climbable Layer

The layers that can be climbed

Start Climb Max Angle

Start climbing when the angle between the character and the climbable object is less than this amount

Start Climb Max Distance

Start climbing when the distance between the character and the climbable object is less than this amount

Min Move To Target Speed

The normalized speed to move to the start climbing position

Search Radius

The radius of the SphereCast to search for a climbable object

Cover



The cover ability allows the character to take cover behind other objects. The cover system is a runtime cover system and pre-defined cover points do not need to be setup ahead of time. Instead, the cover system uses layers to determine if an object can be used for cover. The ability can specify what layers can be used for cover by the CoverLayer LayerMask. When the character initially decides to take cover they will move towards the cover point.

Once arrived at the cover point the character will rotate to face in the opposite direction of the cover. It does this using a raycast at the center of the character's body. As the character strafes while in cover a raycast will be used to keep the character rotated in the opposite direction of the cover.

When the character reaches an edge they will no longer be able to strafe. If the character starts to aim they will pop out of cover ready to fire. As soon as the character is no longer aiming they will move back into cover.

Take Cover Distance

The maximum amount of distance that they can take cover from

Cover Layer

The layers which the character can take cover on

Min Move To Target Speed

The normalized speed that the character moves towards the cover point

Take Cover Rotation Speed

The speed that the character can rotate while taking cover

Cover Offset

The offset between the cover point and the point that the character should take cover at

Cover Angle Threshold

Can move and continue to take cover behind objects as long as the new cover has a normal angle difference less than this amount

Damage Visualization

The Damage Visualization ability allows the character to react when getting hit by an object (bullet, rocket, bassetball bat, etc). This ability will automatically start when the character is damaged. It uses the Animator Controller's additive layer to play its animations.

Fall

The Fall ability allows the character to play a falling animation when the character has a negative y velocity and is not on the ground. The Fall ability is activated automatically and cannot start with an input button.

Height Change

The HeightChange ability allows the character to toggle between height changes. This includes a crouching or crawling state. The Height Change ability has the option of being a concurrent ability so it can be run while other abilities are active. When the ability is active the character's capsule collider will change heights to match the height of the character. This value is specified from the Float Curve Data Animator parameter. This value can be adjusted by changing the [animation curve](#) on the Height Change animations.

Start State

The name of the state when the HeightChange ability starts. Can be blank

Idle State

The name of the state when the character is idle

Moving State

The name of the state when the character is moving

Stop State

The name of the state when the HeightChange ability stops. Can be blank

Concurrent Ability

Is the ability a concurrent ability?

Can Have Item Equipped

Can the item be equipped while the ability is active?

Can Use Upper Body IK

Can the upper body IK be used?

Ledge Strafe

The LedgeStrafe ability allows the character to place their back against the wall and strafe left or right. During this time the character is not able to use their item. This ability is ideal for strafing along a high cliff and the character doesn't want to fall.

The character searches for a ledge to be able to strafe on by the StrafeLayer. In order for the ability to start the character must be a maximum of StartStrafeDistance away from the object. As soon as the ability is activated the character will move to the strafing point at a minimum normalized speed of Min Move To Target Speed and have a offset of StrafeOffset. The character will then continue to be able to strafe along the ledge as long as the ledge angle is less then StrafeAngleThreshold. The StrafeRotationSpeed value specifies how quickly the character can rotate while strafing.

Start Strafe Distance

The maximum amount of distance that the character can start to strafe from

Strafe Layer

The layers that can be used to strafe on

Min Move To Target Speed

The normalized speed to move to the strafe point

Strafe Offset

The offset between the strafe point and the point that the character should strafe

Strafe Angle Threshold

Can move and continue to strafe behind objects as long as the new strafe object has a normal angle difference less than this amount

Strafe Rotation Speed

The speed that the character can rotate while strafing

Interact

The Interact ability allows the character to interact with another object. The object that the character interacts with must implement the [IInteractable interface](#). The Interact ability will show a GUI icon when an interaction can take place.

Interact Target ID

The ID of the IInteractable object that the character can interact with. -1 indicates any

Lower Body State Name

The name of the lower body interact state. Can be empty

Upper Body State Name

The name of the upper body interact state. Can be empty

Min Move To Target Speed

The normalized speed that the character moves towards the target interact position

Can Have Item Equipped

Can the item be equipped while interacting?

Jump

The Jump ability allows the character to jump into the air. Jump is only active when the character has a positive y velocity. When the character has a negative y velocity the [Fall ability](#) starts.

Jump Force

The amount of force that should be applied when the character jumps

Jump Recurrence Delay

Prevent the character from jumping too quickly after jumping

Run Cycle Leg Offset

Determines the correct leg to jump off of

Push

The Push ability allows the character to push other objects. These objects must have the [PushableObject component](#) attached and use the layer specified by the PushableLayer. The character will be able to push the object in the direction that they are facing. In order to push in a different direction the character must change to another side of the object.

In order to start pushing, the character must have an angle less than the StartPushMaxLookAngle and a distance less than StartPushMaxDistance. When the Push ability is activated the character will move to the push position at a minimum normalized speed of Min Move To Target Speed. This push position is specified by the ArmLength value to give a small offset from the character's position. The character will then push with a force of PushForce.

Pushable Layer

The layers that can be pushed

Start Push Max Look Angle

Start pushing when the angle between the character and the pushable object is less than this amount

Start Push Max Distance

Start pushing when the distance between the character and the pushable object is less than this amount

Min Move To Target Speed

The normalized speed that the character moves towards the push point

Arm Length

The length of the character's arms

Push Force

The amount of force to push with

Roll

The Roll ability allows the character to roll. The character can only roll when moving in a relative positive z velocity.

Roll Recurrence Delay

Prevent the character from rolling too quickly after rolling

Speed Change

The SpeedChange ability allows the character to move at a slower or faster rate. This ability is used to add a run or a sprint to the character. Optionally a stamina can be used to prevent character from sprinting too long.

If your character uses root motion the SpeedChange ability will not directly multiply the speed by the Speed Change Multiplier. It will instead multiply the Forward and Horizontal Input parameters belonging to the Animator Controller. This allows the movement blend tree to change the speed by playing a slower or faster movement animation.

Speed Change Multiplier

The speed multiplier when the ability is active

Can Aim

Can the ability be active while the character is aiming?

Use Stamina

Should the character have stamina while in a different speed?

Max Stamina

The amount of stamina the character has

Stamina Decrease Rate

The rate at which the stamina decreases while in a different speed

Stamina Increase Rate

The rate at which the stamina increases while not in a different speed

Swim

The Swim ability allows the character to swim while above water. The Swim ability activates as soon as the character enters the water even though they may not be swimming yet. It will allow normal character movement until the water reaches a predefined depth.

Water Resistance

The amount of resistance to apply while moving

Rotation Speed

The speed that the character can rotate

Swim Depth

The water depth to start swimming at

Swim Camera Offset

The vertical camera offset while swimming

Transition Grace Period

The amount of time that has to elapse before the character can transition between starting and stopping swimming again

Left Hand Splash Particle

Reference to the splash ParticleSystem on the left hand (optional)

Right Hand Splash Particle

Reference to the splash ParticleSystem on the right hand (optional)

Left Foot Splash Particle

Reference to the splash ParticleSystem on the left foot (optional)

Right Foot Splash Particle

Reference to the splash ParticleSystem on the right foot (optional)

Vault

The Vault ability allows the character to vault over other objects. The character can vault over any object specified by the VaultLayer. This object can have a maximum height of MaxVaultHeight and depth of MaxVaultDepth. In addition, this object can be no further away from the character than the distance specified by MoveToVaultDistance.

When the character decides to vault, the character will move up to the VaultOffset at a normalized speed of MoveToVaultSpeed. The actual vaulting animation will then start. This animation uses [Target Matching](#) to ensure the characters hands and feet are placed at the top of the vaulting object. This prevents the characters limbs from clipping with the object or being too high up in the air. The MatchTargetOffset value specifies the offset between the characters hands and the top of the vaulting object. This value is used if the targeting matching doesn't completely align the hands to the vaulting object. StartMatchTarget and StopMatchTarget specify when the character has their hands on the vaulting object and target matching should be active.

Move To Vault Distance

The maximum amount of distance that the character can start moving towards the vault position

Vault Layer

The layers which the character can take vault from

Max Vault Height

The maximum height of the object that the character can vault from

Max Vault Depth

The maximum depth of the object that the character can vault from

Min Move To Target Speed

The normalized speed that the character moves towards the vault point

Vault Offset

The offset between the vault point and the point that the character should start to vault at

Match Target Offset

The offset between the vault point and the point that the character places their hands

Start Match Target

The normalized starting value of the character pushing off the vaulting object

Stop Match Target

The normalized ending value of the character pushing off the vaulting object

Animator Monitor

Unity's Animator Controller has a reputation of quickly becoming a spaghetti mess of transitions. With normal Animator Controller designs, you have a transition from one state to every other possible state that the Animator can play next. In addition, the number of necessary parameters grows in order to be able to trigger the transitions. The Third Person Controller takes a different approach to this controller and transitions only need to be specified when they are absolutely necessary.

The AnimatorMonitor component manages these transitions and will manually transition to new states when requested. This transition occurs using [Animator.Crossfade](#) so no explicit transitions are needed within the Animator Controller. This crossfade transition is generally used when transitioning between groups of animations. For example, the crossfade transition is used when transitioning from an ability back to the normal idle animation. The crossfade transition is not used when transition within a group of animations. For example, explicit transitions are used while within the [Cover ability's](#) substate. Explicit transitions are used in this case because not many transitions are needed and it allows for a more familiar design process.

When using the crossfade transitions, there are two different locations in which the destination state and transition duration are specified. If transitioning to an [Ability](#), the destination state and transition duration is specified within the ability. The AnimatorMonitor contains the destination state and transition duration for all other transitions.

With most Animator Controller designs the transition takes place because a parameter with a specific name is triggered. For example, the character may jump when the Jump bool parameter is true. This results in many, many different parameters if your character can perform many different types of animations. The Third Person Controller framework uses a generic set of parameters that can handle an infinite amount of animations. The parameter list is specified at the bottom of this page.

In addition to managing the transitions, the AnimatorMonitor component is the central point for [animation event](#) callbacks. The AnimatorMonitor will be called when an animation event occurs and the AnimatorMonitor will convert the animation event into an event understood by the rest of the framework using and [EventHandler](#).

Horizontal Input Damp Time

The horizontal input dampening time

Vertical Input Damp Time

The vertical input dampening time

Lower Body Main Substate

The main lower body substate that holds all of the non-ability states

Moving State

The lower and upper body moving state

Moving Aim State

The lower and upper body moving aim state

Aim State

The upper body aim state

Equip State

The state when equipping an item

Unequip State

The state when unequipping an item

Item Use State

The state when using an item

Item Reload State

The state when reloading an item

Secondary Use State

The state when using a secondary item

[Animation Events](#)

Animation Events

When the AnimatorMonitor component receives an [animation event](#) it will convert that animation event into an [EventHandler](#). In most cases the ExecuteEvent or ExecuteEventCheckUpperState animation events can be used. Both of these events take a single string parameter that is used to specify the EventHandler event name. The other events are used when a string cannot be specified. The following animation events are called from the included animations:

ExecuteEvent

Executes an event on the EventHandler

ExecuteEventCheckUpperState

Executes an event on the EventHandler and determine the upper body state

FootDown

The Animator has placed a foot on the ground so a footstep should play

StartAim

The Animator has moved the arms into the aim position

ItemUsed

An item has been used

AlignWithCover

The Animator has moved the character into/out of position to take cover

PopFromCover

The Animator has popped from cover or returned from a pop

IK



The CharacterIK component will rotate and position the character's limbs to face in the correct direction and stand on uneven surfaces. CharacterIK uses Unity's IK solution which requires Unity 5 or Unity 4 pro.

The CharacterIK component will rotate the upper body to look in the direction that the camera is facing. The body, head, and eyes can be influenced differently depending on how much weight each body part should have in looking at the target. CharacterIK will also position the character's feet to match the direction of the surface below them.

Besides positioning limbs, the CharacterIK component also positions of the character's hips. This is used to lower the hip position when the character is standing on uneven ground. As an example, imagine that the character is standing on a set of stairs. The stairs has two sets of colliders: one collider which covers each step, and another collider is a plane at the same slope as the stairs. When the character is standing on top of the stairs, the character's collider is going to be resting on the plane collider while the IK system will be trying to ensure the feet are resting on the stairs collider. In some cases the plane collider may be relatively far above the stair collider so the hip needs to be moved down to allow the character's foot to hit the stair collider.

Debug Draw Look Ray

Draw a debug line to see the direction that the character is facing

Hips Adjustment Speed

The speed at which the hips adjusts vertically

Look Ahead Distance

The distance to look ahead

Look At Offset

An offset to apply to the look at position

Look At Aim Body Weight

(0-1) determines how much the body is involved in the look at while aiming

Look At Body Weight

(0-1) determines how much the body is involved in the look at

Look At Head Weight

(0-1) determines how much the head is involved in the look at

Look At Eyes Weight

(0-1) determines how much the eyes are involved in the look at

Look At Clamp Weight

(0-1) 0.0 means the character is completely unrestrained in motion, 1.0 means the character motion completely clamped (look at becomes impossible)

Look AT IK Adjustment Speed

The speed at which the look at position should adjust between using IK and not using IK

Hand IK Weight

(0-1) determines how much the hands look at the target

Hand IK Adjustment Speed

The speed at which the hand position/rotation should adjust between using IK and not using IK

Foot Position Adjustment Speed

The speed at which the foot position should adjust between using IK and not using IK

Foot Rotation Adjustment Speed

The speed at which the foot rotation should adjust between using IK and not using IK

Foot Weight Adjustment Speed

The speed at which the foot weight should adjust

Footsteps

The CharacterFootsteps component will use animation events to play a footstep when the character moves. An AudioSource must be added to the foot GameObjects to allow the sound to be played from the correct location. A random footstep sound will be played from the Footsteps array.

Feet

A reference to the feet which contain an AudioSource

Footsteps

A list of clips to play when the foot hits the ground

Health



The Character Health component extends on the [Health component](#) by allowing the character to take fall damage. The amount of damage is specified by a curve. The Character Health component listens for the OnControllerLand event and will apply a fall damage if the fall height is greater than MinFallDamageHeight. If the fall is greater than DeathHeight then the character will immediately die. The MinFallDamage and MaxFallDamage is used by the DamageCurve to determine how much damage to apply to the character if the fall height is between MinFallDamageHeight and DeathHeight.

Use Fall Damage

Should fall damage be applied?

Min Fall Damage Height

The minimum height that the character has to fall in order for any damage to be applied

Death Height

A fall greater than this value is an instant death

Min Fall Damage

The amount of damage to apply when the player falls by the minimum fall damage height

Max Fall Damage

The amount of damage to apply when the player falls just less than the death height

Damage Curve

A curve specifying the amount of damage to apply if the character falls between the min and max fall damage values

Respawner

The Character Respawner component extends the Respawner by allowing the character to spawn in a location determined by the [SpawnSelection](#).

Ragdoll



The CharacterRagdoll component will activate when the character dies and turn the character into a ragdoll to let the physics engine react to the damage. The ragdoll can be created using [Unity's Ragdoll Wizard](#). When the character is alive the ragdoll colliders will be disabled and the Rigidbodies will be kinematic. When the character dies the ragdoll colliders will be enabled and the Rigidbodies will be able to freely move. CharacterRagdoll will disable the ragdoll again as soon as the character either spawns or the ragdoll Transforms have settled according to a SettledThreshold.

Settled Threshold

The ragdoll's rigidbodies will be set to kinematic as soon as the transforms have moved a total difference less than this amount between two frames

Spawning

The SpawnSelection component will randomly select a new spawn location out of the SpawnLocations array. SpawnLocations is an array of Transforms and is placed ahead of time while editing the scene. When placing the spawn Transforms, both the position and the rotation of that Transform matter. The object spawned will be facing in the same direction as the spawn Transform. The [Character Respawner component](#) will use the Spawn Selection component in order to determine which location to respawn the character at when the character dies.

Spawn Locations

The locations that the object can spawn

Items

The Item component is the base class of anything that can be picked up. Items can be used by subscribing to the [IUsableItem interface](#) and reloaded by subscribing to the [IReloadableItem interface](#). The Item class is an abstract class and there is no limit to what is considered an item: weapons, baseballs, chainsaws, frying pans. Items don't have to be used only for offense either, for example a health pack or power up can also be considered an item.

If the Item can be used it will work with the [Inventory](#) to determine how much it can be used. The Inventory identifies the Item by the [ItemType](#). The ItemType should be created before the actual item is created. Every item that the character has a chance of using should be [placed on the character](#) while in edit mode. When the game starts the Inventory will search the character for any Items and deactivate any items that the character does not start out with. When the character picks up a new Item it will activate the Item and appear as if the character just picked up the Item.

Items can quickly be created with the [Item Builder](#).

[Shootable Weapon](#)

[Melee Weapon](#)

[Throwable Item](#)

[Attachments](#)

- [Flashlight](#)
- [Laser Sight](#)

Shootable Weapon



The ShootableWeapon component inherits from the Weapon component and allows the weapon to be shot. The ShootableWeapon doesn't have to be used with just bullets, it can be used with anything that is shot. This includes rockets, bow and arrows, pumpkins, etc. ShootableWeapons have many parameters and can keep firing until the trigger is released or are out of ammo. The ShootableWeapon works with the Inventory to determine how much ammo it has.

When the ShootableWeapon is fired it can either fire a projectile or do a hitscan fire (a raycast). If a hitscan fire is used then the weapon will use the HitscanDamageAmount and HitscanImpactForce to determine how much to damage the object and how much force to apply. Hitscans can also spawn decals, dust, sparks, and an impact sound. If the ShootableWeapon fires a Projectile then the projectile applies the damage and force.

If the ShootableWeapon is fired continuously and Overheat is enabled then the ShootableWeapon will overheat for a duration of CooldownDuration. During this time the ShootableWeapon cannot be fired.

In addition, when the weapon is fired it can optionally spawn a Shell, MuzzleFlash, Smoke, and make a sound. When the weapon is out of ammo it will AutoReload if the option is enabled.

ShootableWeapon Items can optionally add the [flashlight](#) or [laser sight](#) attachments.

Fire Point

The point at which to do the actual firing

Fire Mode

The mode in which the weapon fires multiple shots. Options include SemiAuto, FullAuto, and Burst

Fire Rate

The number of shots per second

Burst Rate

If using the Burst FireMode, specifies the number of bursts the weapon can fire

Fire Range

The distance in which the bullet can hit. This is only used for weapons that do not have a projectile

Clip Size

The number of rounds in the clip

Fire Count

The number of rounds to fire in a single shot

Auto Reload

Should the weapon reload automatically once it is out of ammo?

Reload Duration

Should the weapon reload automatically once it is out of ammo?

Recoil Amount

The amount of recoil to apply when the weapon is fired

Spread

The random spread of the bullets once they are fired

Idle Animation State Name

The name of the state to play when idle. The Animator component must exist on the item GameObject

Fire Animation State Name

The name of the state to play when firing. The Animator component must exist on the item GameObject

Reload Animation State Name

The name of the state to play when reloading. The Animator component must exist on the item GameObject

Shell

Optionally specify a shell that should be spawned when the weapon is fired

Shell Location

If Shell is specified, the location is the position and rotation that the shell spawns at

Shell Force

If Shell is specified, the force is the amount of force applied to the shell when it spawns

Shell Torque

If Shell is specified, the force is the amount of torque applied to the shell when it spawns

Muzzle Flash

Optionally specify a muzzle flash that should appear when the weapon is fired

Muzzle Flash Location

If Muzzle Flash is specified, the location is the position and rotation that the muzzle flash spawns at

Smoke

Optionally specify any smoke that should appear when the weapon is fired

Smoke Location

If Smoke is specified, the location is the position and rotation that the smoke spawns at

Fire Sound

Optionally specify a sound that should play when the weapon is fired

Fire Sound Delay

If Fire Sound is specified, play the sound after the specified delay

Empty Fire Sound

Optionally specify a sound that should play when the weapon is fired and out of ammo

Projectile

Optionally specify a projectile that the weapon should use

Projectile Always Visible

Should the projectile always be visible? This only applies to weapons that have a projectile

Projectile Rest Location

If a projectile and always visible is enabled, the location is the position and rotation that the rest projectile spawns at

Projectile Rest Parent

If a projectile and always visible is enabled, the parent is the GameObject that should be the parent of the projectile

Hitscan Impact Layers

A LayerMask of the layers that can be hit when fired at. This only applies to weapons that do not have a projectile

Hitscan Damage Event

Optionally specify an event to send when the hitscan fire hits a target

Hitscan Damage Amount

The amount of damage done to the object hit. This only applies to weapons that do not have a projectile

Hitsscan Impact Force

How much force is applied to the object hit. This only applies to weapons that do not have a projectile

Hitscan Decal

Optionally specify a decal that should be applied to the object hit. This only applies to weapons that do not have a projectile

Hitscan Dust

Optionally specify any dust that should appear on top of the object hit. This only applies to weapons that do not have a projectile

Hitscan Spark

Optionally specify any sparks that should appear on top of the object hit. This only applies to weapons that do not have a projectile

Hitscan Impact Sound

Optionally specify an impact sound that should play at the point of the object hit. This only applies to weapons that do not have a projectile

Overheat

Should the weapon overheat after firing too many shots?

Overheat Shot Count

The number of shots it takes for the weapon to overheat

Cooldown Duration

The time it takes for the weapon to cooldown after overheating

Flashlight

Optionally specify a flashlight that should shine in the direction of the target

Activate Flashlight On Aim

If the flashlight GameObject is specified, should the flashlight automatically activate when aim or should it manually be toggled?

Laser Sight

Optionally specify a laser sight that should point at the target

Activate Laser Sight On Aim

If the laser sight GameObject is specified, should the laser sight automatically activate when aim or should it manually be toggled?

Disable Crosshairs When Laser Sight Active

If the laser sight GameObject is specified, should the crosshairs be removed from the GUI when the laser sight is active?

Melee Weapon



The `MeleeWeapon` component inherits from the `Weapon` component and allows the weapon to apply damage after playing a melee attack animation. A melee weapon can be anything that uses an object within the character's hand to attack. The `MeleeWeapon` implements to the [IUsableItem interface](#) and registers for the used callback so it knows when to apply the melee damage and force.

When a melee attack is started it will wait until the `Used` method is called. The `Used` method is called when an animation event is fired at the attack point of the melee animation. It is at this time that a raycast will be fired to determine what target was hit by the attack. If a target was hit a damage and impact force will be applied to the target.

Attack Rate

The number of melee attacks per second

Attack Distance

The maximum distance that an object can be for it to be attacked

Attack Radius

The radius of the `SphereCast` when determining if a target was hit

Attack Layer

The layers that the melee attack can hit

Attack Sound

Optionally specify a sound that should play when the weapon is attacked

Attack Sound Delay

If Attack Sound is specified, play the sound after the specified delay

Damage Event

Optionally specify an event to send to the object hit on damage

Damage Amount

The amount of damage done to the object hit

Impact Force

How much force is applied to the object hit

Dust

Optionally specify any dust that should appear on at the location of the object hit

Impact Sound

Optionally specify an impact sound that should play at the point of the object hit

Throwable Item



The ThrowableItem component gives the character the ability to throw an item. Any item can be thrown such as grenades, baseballs, water balloons, etc. The GameObject that the ThrowableItem component attaches to is not the actual object that is thrown – the ThrownObject variable specifies this instead. The ThrowableItem component works with the Inventory to determine how many items can be thrown.

The ThrowableItem component implements to the [IUsableItem interface](#) and will wait until the Used method is called before actually letting go of the object. The Used method is called when an animation event is fired at the throw point of the throw animation. It is at this point that the ThrowableItem will spawn the ThrownObject. This ThrownObject must implement the [IThrownObject interface](#) so a force can be applied to the object.

Thrown Object

The object that can be thrown. Must have a component that implements IThrownObject

Throw Rate

The number of objects that can be thrown per second

Throw Force

The force applied to the object thrown

Throw Torque

The torque applied to the object thrown

Spread

A random spread to allow some inconsistency in each throw

Attachments

Attachments extend the functionality of an Item. The following attachments are available:

[Flashlight](#)

[Laser Sight](#)

Flashlight



The flashlight is an optional attachment that can be added to the [ShootableWeapon](#) component. The flashlight can be triggered manually through the ItemHandler or automatically turn on when the character aims.

Laser Sight



The laser sight is an optional attachment that can be added to the [ShootableWeapon](#) component. The laser sight can be triggered manually through the ItemHandler or automatically turn on when the character aims.

Scroll Speed

The speed at which the laser texture scrolls

Max Length

The maximum length of the laser

Min Width

The minimum width of the laser

Max Width

The maximum width of the laser

Pulse Speed

The speed at which the laser changes width

Inventory

The Inventory component manages all of the [Items](#). It allows Items to be used, reloaded, dropped, etc. It also communicates with the Animator to trigger animations when switching between items. The Inventory uses an [ItemType](#) to map an Inventory Item to a particular Item that is held by the character. When the Inventory is initialized it will search through the components on the character for any component that has a type of Item. When it finds an Item it will add that Item's ItemType to the inventory. The character doesn't actually have the Item yet - the Item still needs to be picked up by the character. It is only then that the Item can actually be used. The Inventory has a DefaultLoadout which will load the specified Items when the Inventory is first initialized.

The number of Items in the Inventory can be obtained from the GetItemCount method. This method takes two parameters - the first being the ItemType and the second being if the calling function is interested in the loaded or unloaded count. [PrimaryItems](#) have two different counts: one for the number of [ConsumableItems](#) that are loaded and one for the number of ConsumableItems that are not loaded yet. As an example, the ShootableWeapon uses these two different values for the number of bullets that have been loaded in the weapon and the number of ConsumableItems that have yet to be loaded. When the ShootableWeapon is reloaded it will take the unloaded ConsumableItems and add it to the loaded ConsumableItems.

The GUI interfaces with the Inventory to show the status such as number of bullets remaining or the number of [SecondaryItems](#) left. It does this using the [SharedProperty](#) class.

Item Type

An Item is the base class of any object that can be picked up by the character. The ItemType maps an Item to an object type that the [Inventory](#) can use to update properties on that Item such as if the Item is equipped or how many bullets are left.

Item types are .asset files which can be created using the [Item Type Builder](#). The ItemType is extended by the following classes:

[Primary](#)

[Secondary](#)

[Consumable](#)

[Dual Wield](#)

Primary

The PrimaryItemType extends the [ItemBaseType](#) and is any item type that can be equipped by the character. This includes a standard weapon such as a pistol or shotgun but it can also include throwable items (such as grenade) or melee weapons (such as a knife). The PrimaryItemType uses the [ConsumableItemType](#) to determine how many resources it has left. When an Item is used (such as a weapon being fired), the PrimaryItem count in the Inventory doesn't get decremented. Instead, it is the number of ConsumableItems the PrimaryItem has left.

The PrimaryItemType is used by the Animator to determine which substate to be in. Every Item can have a set of unique animations and the UpperAnimatorSubstate value specifies the name of the substate machine that contain these unique animations. As an example, a one handed item is going to have a different set of run animations compared to a two handed weapon. The LowerAnimatorUseState specifies the state that the lower body Animator should be in when the item is being used. This field will be left empty for most Items.

The UI also uses the PrimaryItemType to show the equipped item's crosshairs and image within the UI. The Crosshairs field specifies the crosshairs and theItemImage field specifies the image that appears on the UI showing that the current weapon is equipped.

Item Name

The name of the item. Will be used by the Animator

Consumable Item

The consumable item that can be used by the primary item

Upper Animator Use State

The upper body substate that the Animator should be in

Crosshairs

The Item crosshairs used by the GUI

Item Sprite

The Item image used by the UI

Secondary

The `SecondaryItemType` extends the [ItemBaseType](#) and is any item type that does not need to be equipped. This includes grenades, health packs, power ups, etc. `SecondaryItemTypes` can be assigned to the same Items that the [PrimaryItemTypes](#) are assigned to, it's just that these Items will never be equipped. `SecondaryItemTypes` specify a capacity since they do not use [ConsumableItemTypes](#) when being used. In addition to the capacity, the `ItemSprite` specifies the UI image that should show indicating the capacity of the `SecondaryItemType`.

Capacity

The maximum number of secondary items the inventory can hold

Item Sprite

The Item sprite used by the UI

Consumable

The `ConsumableItemType` extends the [ItemBaseType](#) and is any item type that can be consumed by the [PrimaryItemType](#). These item types cannot be equipped or used independently. Examples include bullet, flame from a flame thrower, number of melee attacks, etc.

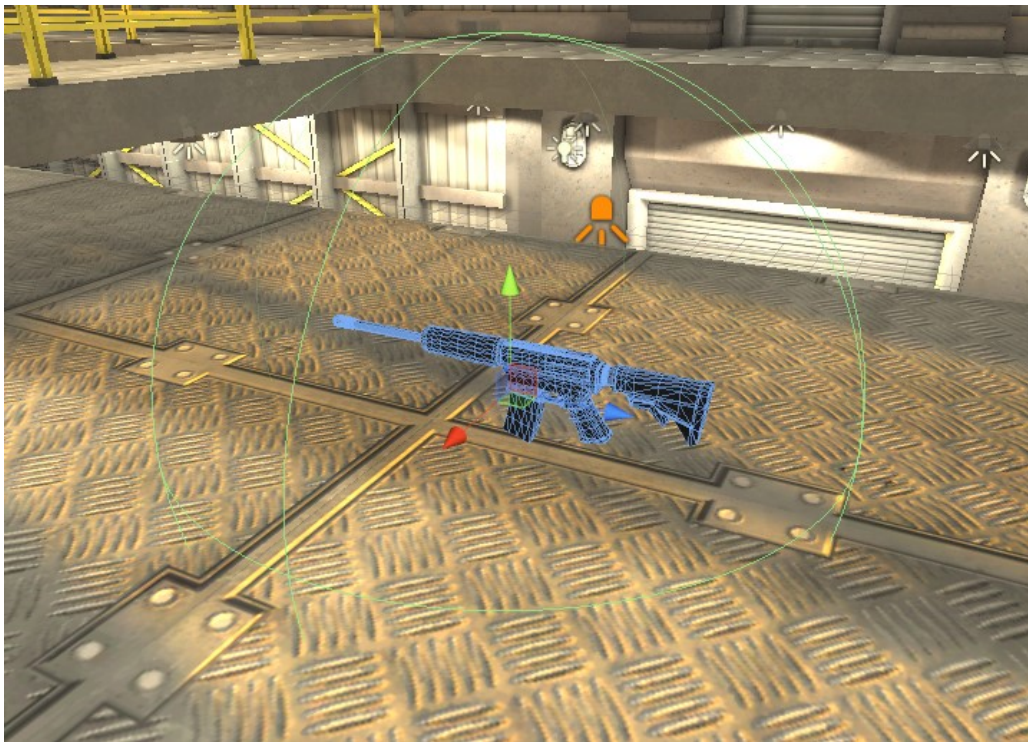
Dual Wield

A dual wield item is any item that can be equipped in conjunction with another primary item of the same type. Examples include dual pistols. If a pistol and a shield can be equipped at the same time, they each will have their own primary type. The shield does not need to use the dual wield item type because two shields are not equipped.

Primary Item

The `PrimaryItem` that this `ItemType` can be equipped with

Item Pickup



The `ItemPickup` component will allow your character to pickup an Item within the scene. In order for the character to pickup an Item, that Item must first be [added to the character](#). Once that Item has been added, the character will then pickup the item when the character enters the `ItemPickup`'s trigger. The `ItemPickup` component uses the [Item Type](#) in order to know which Item to add. Multiple `ItemTypes` can be specified within a single `ItemPickup` component.

Item List

Allows an object with the Inventory component to pickup items when that object enters the trigger

Objects

The following components are considered small objects that don't belong in any other category:

[Climbable Object](#)

[Destructable](#)

- [Grenade](#)
- [Projectile](#)

[Pushable Object](#)

[Shell](#)

[Muzzle Flash](#)

[Explosion](#)

[Moving Platform](#)

Climbable Object

The ClimbableObject component is added to any object in which the character can climb. This includes ladders, vines, or pipes. Even though each type of ClimbableObject has its own traits they all can use the same object. The character can only move up and down vertically while on the ladder. The vines allows the character to move up, down, left, or right as long as the character is not near an edge. When the character is climbing the pipe they must follow the pipe's path. The pipe can move vertically or horizontally.

Climbable Type

The type of object that this component has been added to

Can Reverse Mount

Used by all: can the character mount and face the opposite side of the ClimbableObject?

Bottom Mount Offset

Used by all: the offset that the character should move to when starting to climb from the bottom. A value of -1 means no movement on that axis

Top Mount Offset

Used by ladders and vines: the offset that the character should move to when starting to climb from the top. A value of -1 means no movement on that axis

Rung Separation

Used by ladders: the separation between ladder rungs

Unusable Top Rungs

Used by ladders: the number of rungs that are unusable by the characters feet

Horizontal Padding

Used by vines: the padding from the left and right sides of the box collider that prevent the character from moving too far horizontally

Top Dismount Offset

Used by vines: the offset from the top of the box collider that the character should start dismounting from

Bottom Dismount Offset

Used by vines and pipes: the offset from the bottom of the object that the character should start dismounting from

Mount Positions

Used by pipes: the transforms of the positions that the character can start climbing from

Horizontal Transition Offset

Used by pipes: the horizontal offset from the object position to transition from a vertical to horizontal pipe

Vertical Transition Offset

Used by pipes: the vertical offset from the object position to transition from a horizontal to vertical pipe

Extra Forward Distance

Used by pipes: the amount of extra transition distance of the forward movement compared to the backward movement

Destructable

The Destructable component is an abstract class which acts as the base class for any object that immediately destroys itself and applies a damage. The [Projectile](#) and [Grenade](#) components are inherited from the Destructable class. When the object is destroyed it can spawn an explosion, decal, or dust prefab.

Damage Amount

The amount of damage done to the object hit. Will not be used if an explosion is specified

Impact Force

How much force is applied to the object hit. Will not be used if an explosion is specified

Damage Event

Optionally specify an event to send when the object is hit. Will not be used if an explosion is specified

Explosion

Optionally specify an explosion prefab

Decal

Optionally specify a decal when the destructable activates

Dust

Optionally specify dust when the destructable activates

Grenade



The Grenade component extends the [Destructable component](#) to allow an object to be destroyed after the specified Lifespan. When this Lifespan has elapsed the parent Destruction method will be called to do the actual destruction. To start this timer the Initialize method must be called and this method has two parameters: force and torque. These parameters will apply a relative force and torque to the Rigidbody when the Grenade is initialized.

Lifespan

The length of time the Grenade should exist before it destroys itself

Projectile



The Projectile component extends the [Destructable component](#) to allow an object to move using a Rigidbody. The Projectile will keep moving at the specified Speed until it collides with another collider or the Lifespan duration has been reached. When one of these events do happen the Projectile will call the parent Destruction method to do the actual destruction.

Speed

How quickly the Projectile should move

Lifespan

The length of time the Projectile should exist before it activates if no collision occurs

Pushable Object

The PushableObject component is added to any object which can be pushed by the character. The PushableObject requires a Rigidbody and when the object is not being pushed the Rigidbody is kinematic.

Dampening

The amount of dampening force to apply while moving

Shell



The Shell component is added to an object which will fly out of a weapon to help indicate that a bullet has been fired. Because the shell is so small Unity's physics engine doesn't work too well with it so the shell's Rigidbody will be set to kinematic after it has landed on the floor. The shell will stay on the floor for the Lifespan duration and then placed back in the ObjectPool.

Lifespan

The length of time that the shell should exist for

Muzzle Flash



The MuzzleFlash component will slowly fade the Renderer when the component is enabled. The renderer must have the "_TintColor" color defined to be able to apply the fade. A light may optionally be added to the object and that light will be faded at the same rate as the Renderer.

Start Alpha

The alpha value to initialize the muzzle flash material to

Fade Speed

The larger the value the quicker the muzzle flash will fade

Explosion



The Explosion component will apply a force and damage any object that is within the specified Radius. A sound can be played as well to indicate that an explosion took place. The Explosion object will be placed back in the ObjectPool after the Lifespan duration has elapsed. An explosion can occur either when enabled or triggered manually with the Explode method.

Explode On Enable

Should the explosion explode when the object is enabled?

Lifespan

The duration of the explosion

Radius

How far out the explosion affects other objects

Impact Force

The amount of force the explosion applies to other Rigidbody objects

Damage Event

Optionally specify an event to send to the object hit on damage

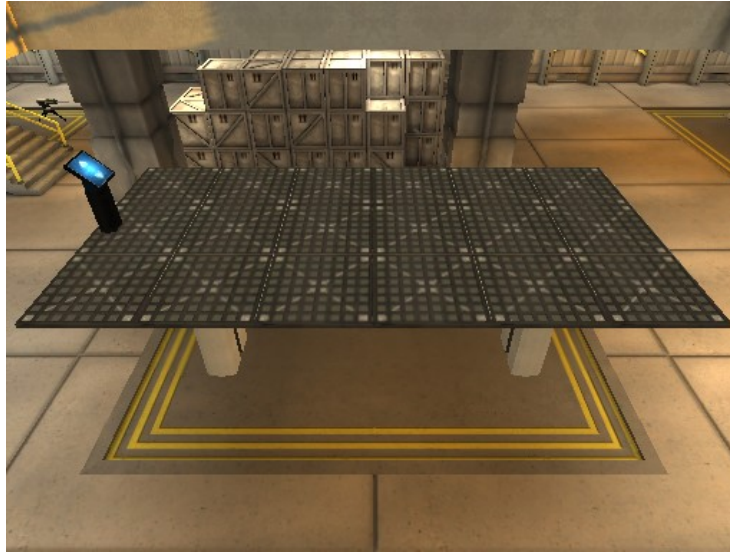
Damage Amount

The amount of damage the explosion applies to other objects with the Health component

Sound

Sound to play during the explosion

Moving Platform



The MovingPlatform component will smoothly move an object between any number of waypoints. In addition to position changes, the platform can change rotations as defined by the RotationSpeed. As soon as the platform arrives at a waypoint it will either smoothly come to a stop or continue on to the next waypoint. The MovingPlatform component implements the [IInteractableTarget interface](#) and will start moving when Interact is called.

The RigidbodyCharacterController component can smoothly ride on top of MovingPlatforms. In order for it to do this it must know that it is on top of a MovingPlatform and this is specified by the [MovingPlatform layer](#).

Move When Enabled

Should the platform start moving when the object is enabled?

Move Dampening

The amount of dampening to make the movement smooth

Rotation Speed

The speed at which the platform rotates

Stop At Waypoint

Should the platform stop moving when it arrives at a waypoint?

Can Interact While Moving

Can the direction be changed while the platform is moving?

Waypoints

An array of Transforms which the moving platform will move towards. When the ending waypoint is reached it will loop back to the start

Start Waypoint Index

The starting waypoint index to move towards

Traits

Traits are components which add small functionality to an object. The following traits are included:

[Health](#)

- [Health Pickup](#)

[Interactable](#)

[Particle Remover](#)
[Respawner](#)

Health

The Health component allows any object to take damage inflicted by another object. This component can be added to any object to give that object health – whether that be a character or an inanimate object such as a barrel. The health component does not require any other components to work.

The Health component has two types of health – standard health that does not regenerate and health that does regenerate (called a shield). Both types of health are not required. For example, an object can only use its shield if it has a MaxHealth value set to zero.

When the health and shield values reach zero the object is considered dead. When the object dies it will spawn any objects in the SpawnedObjectsOnDeath array and deactivate if DeactivateOnDeath is enabled. In addition, the OnDeath and OnDeathDetails events are executed which notifies interested objects that the object has died.

An object can take damage by calling the Damage method. This method takes three parameters: an amount, position, and force. The amount specifies the amount of damage to take. If the object has both health and a shield then the amount will be subtracted from the shield first. The position and force parameters specify where the damage occurred and how much force was applied. This information is used by the GUI for the hit indicators and the ragdoll if the object died and has a ragdoll attached.

Health can be added by collecting the [Health Pickup](#).

Invincible

Is the object invincible? If true no damage will be taken

Max Health

The maximum amount of health, can be 0

Max Shield

The maximum amount of shield, can be 0

Shield Regenerative Initial Wait

If using a shield, the amount of time to initially wait before the shield regenerates

Shield Regenerative Amount

If using a shield, the amount to regenerate every interval

Shield Regenerative Wait

If using a shield, the amount of time to wait before regenerating a small amount of shield

Spawned Objects On Death

Any objects that should spawn when the object dies

Deactivate On Death

Should the object be deactivated on death?

Health Pickup



The HealthPickup component allows the character to pickup health. When the character enters the health pickup trigger, the HealthPickup will immediately increase the object's health by the specified amount. This object must have the [Health](#) component attached.

Heal Amount

The amount of health to add

Interactable



The Interactable component allows the object to be interacted with. Examples of use include switches or buttons. In order for an interaction to occur, the character that will be interacting with the Interactable object must be within the Interactable's trigger and be looking at the Interactable object.

The Interactable object specifies which object it can interact with. This object, which implements the [IInteractableTarget interface](#), can be any object that performs an action. Examples include a moving platform, opening door, or turning on a light.

When the Interactable object is interacted with it will register itself for the OnAnimatorInteracted event. This event is executed by the AnimatorMonitor when the interact animation fires the Interacted animation event. It is only when this event is executed that the Interactable will tell the IInteractableTarget to perform its action. At any point during this time the IInteractableTarget may become unavailable for any reason so immediately before the action is performed the Interactable object asks the IInteractableTarget if it can still be interacted with.

Target

The object perform the interaction on. This object must implement the IInteractableTarget interface

Interactor Look Interact Threshold

The amount that the interactor must be looking at the Interactable in order to interact. -1 is completely looking at the target, 1 is looking in the opposite direction

Camera Look Interact Threshold

The amount that the camera must be looking at the Interactable in order to interact. -1 is completely looking at the target, 1 is looking in the opposite direction

Max Horizontal Offset

The maximum x offset that the character can be standing away from the Interactable in order to interact

Interactor Layer

The layer of objects that can perform the interaction

Particle Remover

The ParticleRemover component attaches to an object which has a ParticleSystem and will place the object back in the ObjectPool when the ParticleSystem is done playing.

Respawner

The Respawner component will spawn the object again after it has been disabled. It will spawn the object after the RespawnTime duration has elapsed. When the object spawns again it will have same position and rotation as it did when the object was first enabled. When the object is spawned again it will execute the OnRespawn event.

Respawn Time

Waits the specified amount of time before respawning

Layers

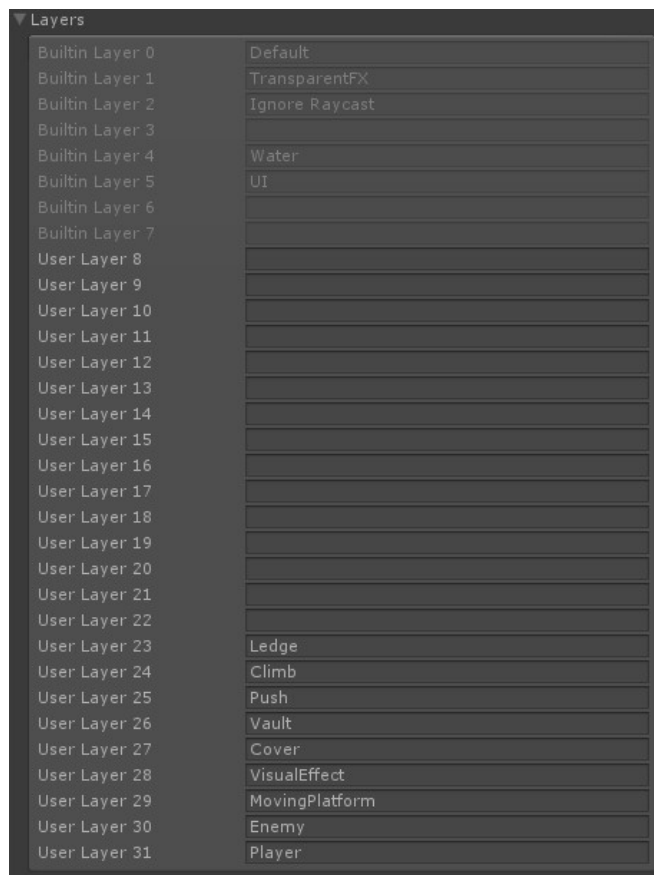
The Third Person Controller is not in the "Complete Projects" category of the Asset Store and therefore does not import any project settings. Custom

layers are however necessary for certain features to work properly such as the [moving platforms](#) or [cover](#). The LayerManager class allows this to happen by defining the layers without the layers having to exist in Unity's TagManager.

The following custom layers are setup:

Ledge – Layer 23
 Climb – Layer 24
 Push – Layer 25
 Vault – Layer 26
 Cover – Layer 27
 VisualEffect – Layer 28
 MovingPlatform – Layer 29
 Enemy – Layer 30
 Player – Layer 31

These layers should be setup manually in the [TagManager](#) for them to be visible on the object which belongs in that layer.



Input

Unity's [InputManager](#) is used for getting input from the player. Keyboard and mouse, controller, and mobile input is supported. The UnityInput component is used as a common base class for both [standalone](#) and [mobile](#) input. To easily allow integration with other [input implementations](#), the UnityInput class is derived from the PlayerInput class. The PlayerInput class is used by all of the calling components in order to determine if the specified input is activated.

Note: When the Third Person Controller is first downloaded from the Asset Store it does not include a custom InputManager file which contains the correct inputs for a third person controller. The InputController can be updated by opening the Third Person Controller Start Window and selecting Update Input. The InputManager will then add any missing inputs.

Standalone

Standalone input is used by any platform that is not a mobile platform. This includes the PC, Mac, and consoles. In most cases the [InputManager](#) maps the controller and keyboard to the same name. The only time this does not occur is for the controller's trigger controls. The triggers are considered an axis whereas with a keyboard and mouse they are regular buttons. In this case the StandaloneInput class first tries to get the controller input and if that fails it will then try to get the input from the keyboard.

Note: When the Third Person Controller is first downloaded from the Asset Store it does not include a custom InputManager file which contains the correct inputs for a third person controller. The InputController can be updated by opening the Third Person Controller Start Window and selecting Update Input. The InputManager will then add any missing inputs.

Mobile



MobileInput does not use the InputManager and instead completely relies on touch input. This touch input can be retrieved with the VirtualButton component. The VirtualButton accepts both presses and swipes. Presses are similar to button presses on the keyboard and swipes are similar to axis input on the keyboard. All of the VirtualButtons should be placed under a common GameObject which contains the VirtualButtonManager. This VirtualButtonManager component will enable or disable the VirtualButtons according to the PlayerInput settings. No extra work is required to get mobile controls to work – the same PlayerInput class can be used to get both standalone and mobile input and most components do not even need to know that the game is running on a mobile device.

Note: for MobileInput to work correctly, make sure you first import the [source code](#).

Component Communication

The components within the Third Person Controller are designed to be very modular. Don't want an object to have health? No problem, don't add the Health component. Or what if you don't want an object to respawn after it gets destroyed? Just don't add the Respawner component.

This modularity is achieved through events, schedules, and property/method sharing. In the case where one component requires another component, such as the CameraController requiring a CameraHandler, the two classes are as loosely coupled as possible. For this camera example, the CameraController is aware of the CameraHandler but the CameraHandler does not know about the CameraController.

Events

The EventHandler classes adds a generic event system to the framework. This event system allows objects to register, unregister, and execute events on particular objects. Multiple objects can subscribe to the same event. The Third Person Controller uses events extensively to allow components to communicate without directly knowing about each other. The following definition is used to listen for an event:

```
EventHandler.RegisterEvent(object obj, string name, Action handler)
```

This function is overloaded to allow events with multiple parameters. Unregistering from an event is very similar as it is to register for an event:

```
EventHandler.UnregisterEvent(object obj, string name, Action handler)
```

An event can then be executed with:

```
EventHandler.ExecuteEvent(object obj, string name)
```

As an example, the "OnRespawn" event is executed when the Respawner component spawns the object again. A component can listen for this event by first registering:

```
EventHandler.RegisterEvent(gameObject, "OnRespawn", RespawnFunction);
```

Where RespawnFunction is the function that should execute when the "OnRespawn" event is executed. This function might look like:

```
private void RespawnFunction()
{
    // Respawn logic
}
```


The GameObject can then unsubscribe itself from the event with the following:

```
EventHandler.UnregisterEvent(gameObject, "OnRespawn", RespawnFunction);
```

The OnRespawn event will be called by the Respawner class by calling ExecuteEvent:

```
EventHandler.ExecuteEvent(gameObject, "OnRespawn");
```

Every object which is subscribed to the "OnRespawn" event of that particular GameObject.

Shared Property and Method

In some cases a component may want to get/set a value or execute a function on a separate component without directly knowing about that separate component. As an example, when a weapon is fired it sets a recoil amount so the recoil can be portrayed on the GUI by expanding or moving the crosshairs. Since the framework works with both player-controlled characters and AI-controlled characters, there isn't always guaranteed to be a GUI for every weapon fired. Furthermore, every game is different so a GUI that displays the recoil may not even be necessary. Instead of requiring the Weapon component to have knowledge that the GUI component exists, a SharedProperty can be used instead. SharedMethod is similar to SharedProperty except it works on methods instead of properties.

The first step to setup a SharedProperty is to first define the variable that holds the property value. From the recoil example it would be a simple float:

```
private float m_RecoilAmount = 0;
```

The property for this variable then needs to be have a special formatting in order for other components to know that this property exists. This special formatting means that the property needs to have "SharedProperty_" appended before the name of the property:

```
private float SharedProperty_RecoilAmount { get { return m_RecoilAmount; } set { m_RecoilAmount = value; } }
```

After the "SharedProperty_" is the name of the variable that the property gets or sets. Do not include the "m_" as that will be ignored. SharedProperties will not work if this formatting is not applied. For example, the following will property name will NOT work:

```
SharedProperty_Recoil
```

This property name will not work because the property doesn't have the full variable name, RecoilAmount.

After the property has been created the component needs to register itself with the SharedManager so other components can find it. This should be done within the Awake method:

```
private void Awake()
{
    SharedManager.Register(this);
}
```

This is all that is required to have the property be recognized. In order for another component to find this property it first needs to have a variable which will be used to store the reference to the property:

```
private SharedProperty m_RecoilAmount = null;
```

Take note that this variable has the same name as the variable that the SharedProperty modifies. This is also necessary as it maps a particular variable to the SharedProperty. The component then needs to initialize the SharedProperty with the correct property reference. This should be done within the Start method:

```
private void Start()
{
    SharedManager.InitializeSharedFields(guiGameObject, this);
}
```

The first parameter is the object that the original SharedProperty has been added to. In this example the object is the GameObject that the GUI has been added to. This is all that is required in order to get the components to share properties without directly knowing about each other!

The RecoilAmount SharedProperty can then be used by calling Get and Set:

```
float amount = m_RecoilAmount.Get();
m_RecoilAmount.Set(amount);
```

SharedMethod works in a very similar way as SharedProperty does except it doesn't require as much setup because no values are being stored. The first step is to create the method:

```
public Vector3 SharedMethod_TargetLookPosition()
{
    // Implementation
}
```

Notice that this method has a very similar format as SharedProperty: it must start with "SharedMethod_" and the name of the method ("TargetLookPosition") is used to map the variable to the method. The component which contains this method still must register itself within Awake:

```
Public void Awake()
{
    SharedManager.Register(this);
}
```

In order for another component to find this method it needs to have a variable which will be used to store the reference to the method:

```
private SharedMethod TargetLookPosition = null;
```

Notice that the name of this variable matches the name of the method after the “SharedMethod_” prefix. This component then needs to initialize the SharedMethod with the correct reference:

```
private void Start()
{
    SharedManager.InitializeSharedFields(targetGameObject, this);
}
```

The same parameters are used with SharedMethod as SharedProperty. The first parameter is the object that the SharedMethod method exists on. Once this is setup the component is able to execute methods without having direct knowledge on what object it is executing these methods on! As an example, this SharedMethod can then be used by calling invoke:

```
TargetLookPosition.Invoke();
```

Scheduler

While not directly related to communicating with other components, the Scheduler allows the delayed execution of methods. When a new event is scheduled, a ScheduledEvent object is created and can then be stored which then allows the event to be cancelled. The following definition is used to schedule an event:

```
ScheduledEvent Schedule(float delay, Action handler)
```

Where delay is the amount of time that should elapse before handler is executed. The scheduler can also accept parameters:

```
ScheduledEvent Schedule(float delay, Action handler, object arg)
```

This ScheduledEvent can later be cancelled with:

```
Cancel(ScheduledEvent scheduledEvent)
```

As an example, the Explode method will be called after 1.5 seconds:

```
Scheduler.Schedule(1.5f, Explode);
```

Object Pool

It is a relatively expensive operation to instantiate and destroy objects. To avoid many instantiations and destructions an ObjectPool is used which will reuse an already-created object. The ObjectPool class can pool both GameObjects and regular objects derived from System.object.

If you are wanting to create a new object, ObjectPool.Instantiate should be used instead of Instantiate. This function will create a new GameObject if one doesn't already exist within the pool. Similarly, to destroy an object, ObjectPool.Destroy should be used instead of Destroy. This will place the GameObject back in the pool.

In a similar way that GameObjects can be pooled, regular System.object objects can be pooled as well. To get a new object, ObjectPool.Get< T >() should be used. T specifies the type of object to get. For example, if you have a class named CustomClass you can get the pooled object with ObjectPool.Get< CustomClass >(). To return an object back to the pool ObjectPool.Return(obj) can be used.

Interfaces

The following interfaces can be subscribed to:

[IInteractable](#)
[IInteractableTarget](#)
[IReloadableItem](#)
[IUsableItem](#)
[IThrownObject](#)
[IFlashlightUsable](#)
[ILaserSightUsable](#)

IInteractable

IInteractable provides an interface for objects that perform the interaction on an [IInteractableTarget](#). Examples of objects that may implement the IInteractable interface are switches or buttons. The [Interactable component](#) implements this interface.

InteractableTarget

InteractableTarget provides an interface for objects that can be interacted with. Examples of objects that may implement the InteractableTarget are platforms, doors, or lights. The [MovingPlatform component](#) implements this interface.

IReloadableItem

IReloadableItem provides an interface for an [Item](#) that can be reloaded. Weapon components implement this interface.

IUsableItem

IUsableItem provides an interface for an [Item](#) that can be used. Examples of being used include being fired, swung, or thrown. The Weapon and ThrowableItem components implement this interface. This interface provides the methods which allow the handler to try to use the Item, determine if the Item is in use, and try to stop the in-use Item.

The implementer can register to receive a callback to know when to actually be used. An example of use for this is when the Grenade is being thrown. The Grenade shouldn't be thrown immediately because it should wait for the hand's animations to get to an appropriate release point. When the hands reach this release point an animation event will be triggered which will then callback to the IUsableItem.

IThrownObject

IThrownObject provides an interface for an object that can be thrown. The [Grenade component](#) implements this interface.

IFlashlightUsable

IFlashlightUsable provides an interface for Items that can add a flashlight attachment. The [ShootableWeapon](#) component implements this interface.

ILaserSightUsable

ILaserSightUsable provides an interface for Items that can add a laser sight attachment. The [ShootableWeapon](#) component implements this interface.

Integrations

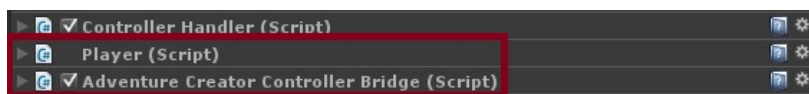
The following topics explain in depth how the Third Person Controller integration is setup:

[Adventure Creator](#)
[Dialogue System](#)
[Final IK](#)
[Input \(Rewired, Control Freak\)](#)
[ORK](#)
[plyGame](#)
[UMA 2](#)

Adventure Creator

The [Adventure Creator](#) integration enables the Third Person Controller character controller to be used with the Adventure Creator framework. The following steps are required to have your character work with Adventure Creator:

1. Create your character like normal through the [Character Builder](#).
2. Add the Adventure Creator Player component.
3. Add the Adventure Creator Controller Bridge component. This can be found on the [Integrations page](#).



No other steps are required. Adventure Creator will take control of the character when it needs to (such as during cutscenes). For all other times the Third Person Controller will have control.

If you are trying the Adventure Creator sample scene you should do the following:

1. Import Adventure Creator.
2. Import the Third Person Controller.
3. Import the Adventure Creator integration assets.
4. Import the Adventure Creator sample scene.

5. Open Third Person Controller/Third Party/Adventure Creator/Sample/scene.unity.
6. Click on Third Person Controller/Third Party/Adventure Creator/Samples/Adventure Creator Sample_ManagerPackage.asset so it opens in the Unity inspector.
7. Click "Assign Managers" within the inspector.
8. Play.

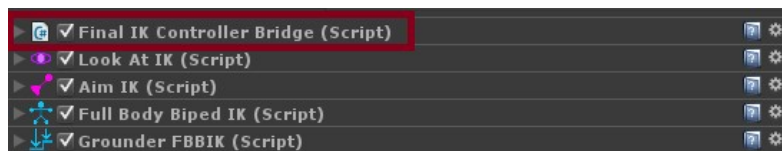
Dialogue System

The [Dialogue System for Unity](#) integration was developed by Pixel Crushers. Please see [this page](#) for its documentation.

Final IK

The [Final IK](#) integration allows your character to use Final IK for its lower and upper body IK instead of Unity's IK system. The integration with Final IK is quick to setup:

1. Ensure the Character IK component is removed from the character. Note that this component may not have been added if you did not have "Add IK" selected within the Character Builder.
2. Add the Final IK Controller Bridge to your character. This can be downloaded from the [Integrations page](#).
3. Add your Final IK components. In the sample scene we added the Look At IK, Aim IK, Full Body Biped IK, and Grounded FBBIK.



Input (Rewired, Control Freak)

The Third Person Controller is integrated with [Rewired](#) and [Control Freak](#) for alternate input implementations. In order to use one of these integrations, the corresponding Input component should be used instead of UnityInput. Ensure the UnityInput component has been removed after adding a third party input integration. For example, if you want to use Rewired, the RewiredInput component should be added and the UnityInput component removed. The Third Person Controller will always use this input implementation instead of Unity's InputManager.

ORK

The [ORK](#) integration enables the Third Person Controller character and camera to be used with the ORK framework. The following steps are required to setup your character to be used with ORK:

1. Create your character like normal through the [Character Builder](#).
2. Add the ORKCharacterControllerBridge component to your character. This can be found on the [integrations page](#).
3. Following the [ORK documentation](#), open the Base/Control -> Game Controls editor within the ORK Framework window.
4. Set the Player Control Type to None.
5. Add a new Custom Control with the following values. The other fields can keep their defaults.
 - Blocked By: Player
 - Placed On: Player
 - Behaviour Name: ORKCharacterControllerBridge
6. Save your settings within the ORK Framework

After completing these steps the Third Person Controller can now be used by ORK. ORK will automatically take control of the character when it needs to (such as during interactions).

The Third Person Controller camera can also be used with ORK. It is setup in a similar way as the character:

1. Add the ORKCameraControllerBridge component to your camera. This can be found on the [integrations page](#).
2. Following the [ORK documentation](#), open the Base/Control -> Game Controls editor within the ORK Framework window.
3. Set the CameraControl Type to None.
4. Add a new Custom Control with the following values. The other fields can keep their defaults.
 - Blocked By: Camera
 - Placed On: Camera
 - Behaviour Name: ORKCameraControllerBridge
5. Save your settings within the ORK Framework

The ORK sample project can be downloaded from the [integrations page](#). Perform the following steps to get this sample working:

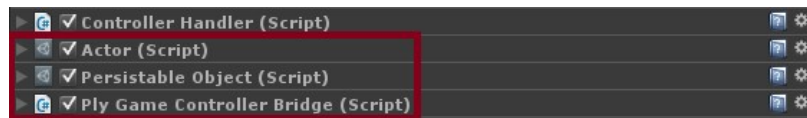
1. Download and extract the ORK demo from the [ORK website](#).
2. Import the Third Person Controller.

3. Update the Input Manager from the Third Person Controller Start Window.
4. Open the ORK demo project.
5. Import the Third Person Controller ORK sample package.
6. Open the scene found in Tutorial Resources/Scenes/0 Main Menu. This scene has been updated with the Third Person Controller integration.
7. Play.

plyGame

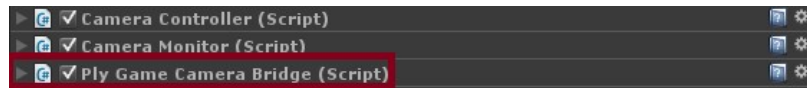
The [plyGame](#) integration enables the Third Person Controller character and camera controller to be used with the plyGame framework. The following steps are required to have your character work with plyGame:

1. Create your character like normal through the [Character Builder](#).
2. Add the plyGame Actor and Persistable Object component.
3. Add the plyGame Game Controller Bridge component. This can be found on the [Integrations page](#).



No other steps are required. With this setup plyGame will be aware of the Third Person Controller character controller. The camera can be setup similarly:

1. Add the Third Person Controller Camera Controller component.
2. Add the plyGame Camera Bridge component
3. Add the plyGame Game Controller Bridge component. This can be found on the [Integrations page](#).

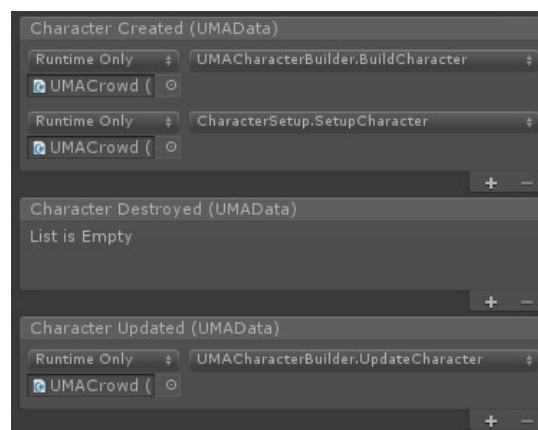


If you are trying the plyGame sample scene you should do the following:

1. Import plyGame.
2. Import the Third Person Controller.
3. Import the plyGame integration assets.
4. Import the plyGamer sample scene.
5. Add the scene located at Third Person Controller/Third Party/plyGame/System/00-bootstrap.unity to the Unity build menu.
6. Open the scene at Third Person Controller/Third Party/plyGame/scene.unity
7. Play.

UMA 2

The [UMA 2](#) integration allows your UMA created character to be used as a Third Person Controller character. There are two different components included in this integration. The first component adds all of the Third Person Controller components after UMA has created the character. These components are the same components that the [Character Builder](#) adds. The second component is game specific and is included in the UMA sample scene download. This component will add the Jump and Fall abilities, along with attaching the camera. Both of these components use [Unity Events](#) to be notified after the UMA character has been created:



This screenshot was taken on the UMA Dynamic Avatar component. When the character is created UMACCharacterBuilder.BuildCharacter will be called. This adds of the standard Third Person Controller components. The second event, CharacterSetup.SetupCharacter, is game specific and in this case will add the Jump and Fall ability along with attaching the camera to the character. When the character is updated UMACCharacterBuilder.UpdateBuilder is

called and this will do any Third Person Controller updates, such as resizing the character's CapsuleCollider.

Support

We are here to help! If you have any questions/problems/suggestions please don't hesitate to ask. You can email us at support@opsive.com or post on the [forum](#).