Character Movement Fundamentals

Rigidbody-based Character Movement

User Manual

Version 1.3

November 20, 2019

Quickstart

Example Scenes

Two example scenes are included in this package to showcase the controller's abilities and possible applications.

Fig. 1 *Folder location of example scenes in the package.*

'ShowcaseScene' is meant to demonstrate how the controller moves on different terrains, slopes and stairs as well as to showcase some interesting gameplay possibilities using special level elements (moving platforms, switching gravity).

It also allows you to switch between different controller prefabs (first person prefab, multiple third person prefabs).

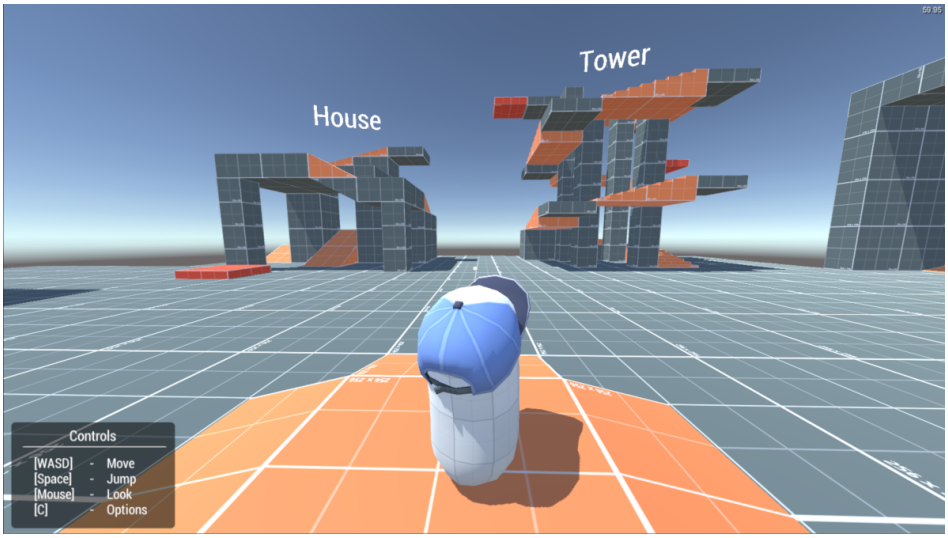


Fig. 2 *Showcase scene.*

'TopDownScene' offers a similar experience, but with a focus on gameplay from a top down perspective.

Just open the scenes, press play and you're ready to go! Player controls are explained in the scenes using UI elements or text meshes.

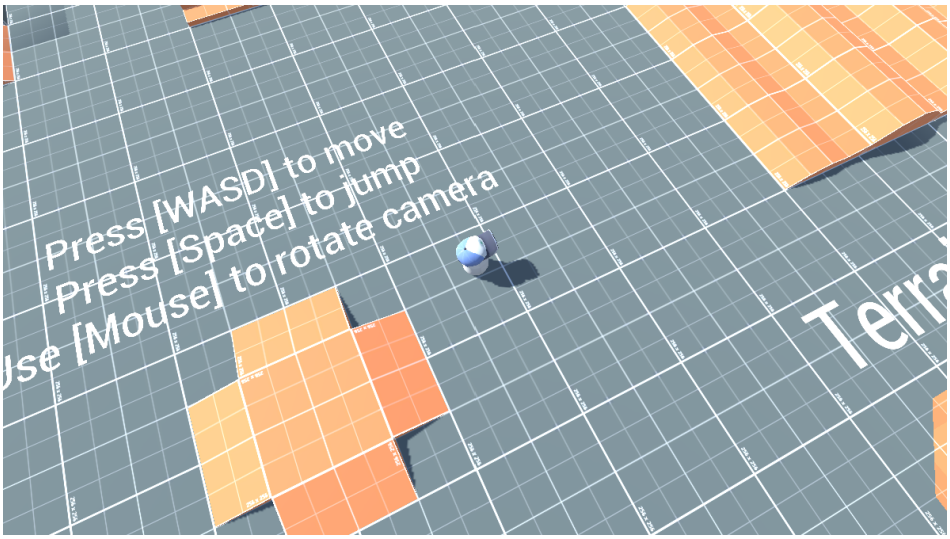


Fig. 3 *Topdown scene.*

Prefabs

This package also comes with 12 fully functional controller prefabs, which can be found in the 'Controller Prefabs' folder. They are organized into *'animated'* and *'blank'* prefabs.

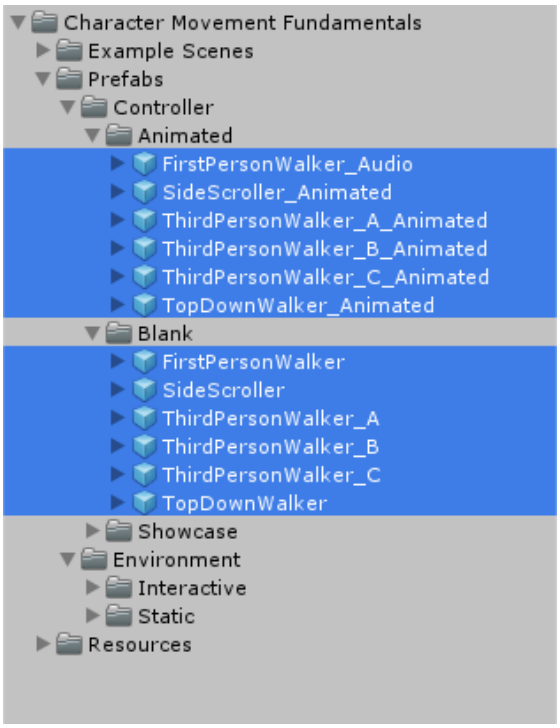


Fig. 4 *Folder location of prefabs in the package.*

Blank controller prefabs are purposely put together using only components that are absolutely necessary for the controller to function. They are meant to be a 'blank slate' or base from which start from.

Animated controllers come with basic animation (and sound) already set up and serve as an example on how to link animations and sound cues (like footsteps) with the controller's movement.

Alternatively, they are also very well suited for quick prototyping.

You can use these prefabs by simply dragging them into an empty scene - no further setup is required.

All controller prefabs come with their own cameras, so make sure to delete any unused cameras in the scene afterwards.

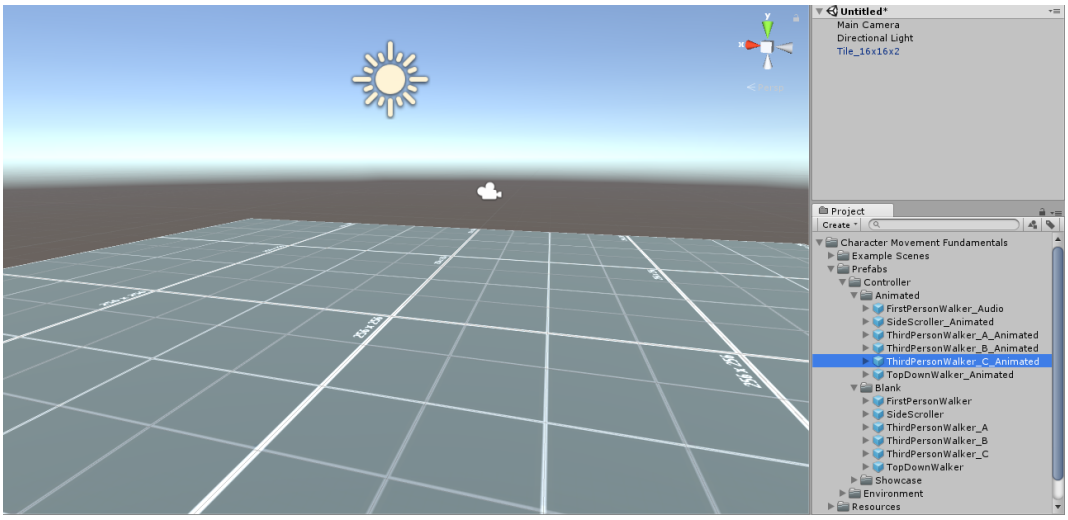


Fig. 5 *Select a controller prefab ...*

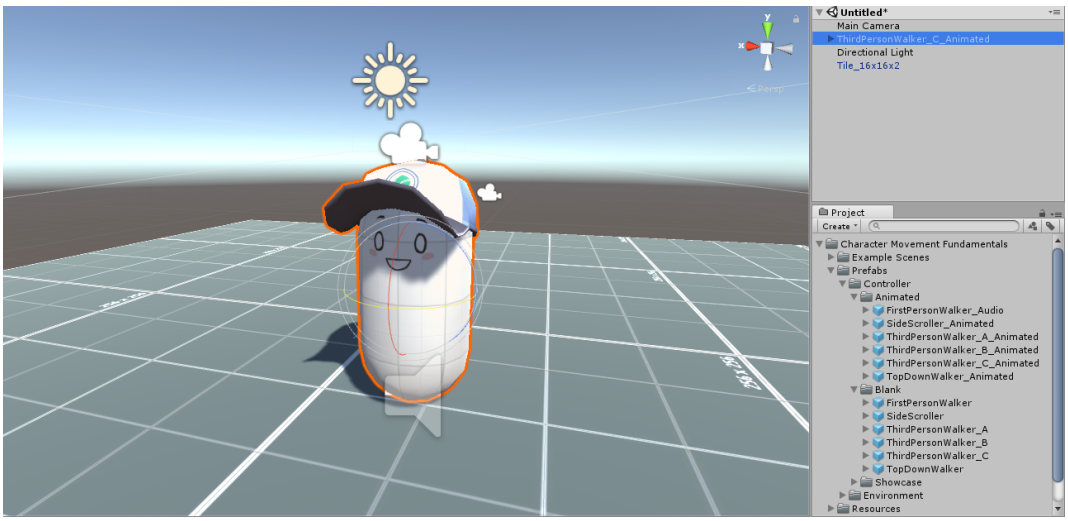


Fig. 6 *... simply drag it into into an empty scene ...*

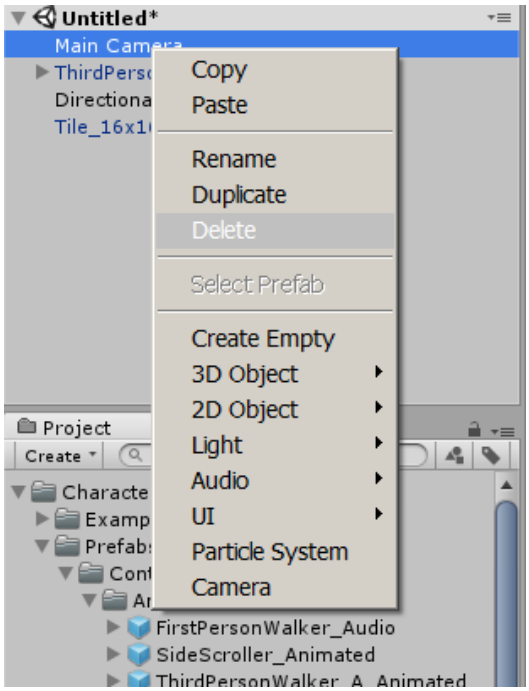


Fig. 7 *... and delete the old camera.*

Contents

1 Introduction

Thank you for your interest in this package! I'm sure it will serve you well and help you realize your game ideas in Unity!

In case you run into any problems or if you need any help with something related to this package, feel free to contact me at ***support@j-ott.com***.

The same goes for any feedback or thoughts you might have about this package, of course! I'd love to hear your ideas!

(Note: Please keep the ***invoice number*** *you received as part of your purchase at hand when requesting support via email! Thanks!)*

2 About this Package

2.1 What Is This?

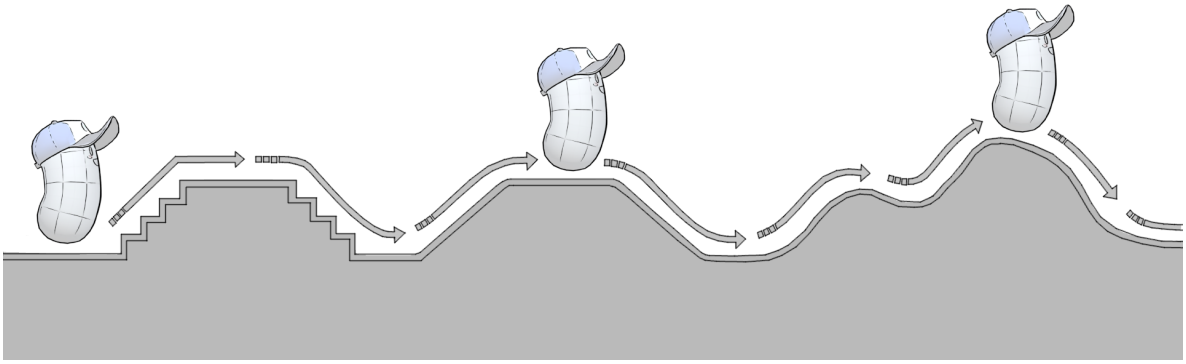


Fig. 8 *Smoothly moving over stairs, slopes and terrain.*

'Character Movement Fundamentals' is a rigidbody-based character movement system.

Put simply, it is a collection of scripts, components and prefabs that will help you quickly set up characters that can *move around* and *react to a game environment* in a *physically believable* way.

It's both *easy to understand* and *easy to adapt* to your game's specific needs, either by modifying the scripts in this package or even writing your own.

I specifically didn't want this package to be a hyper complex, highly specialized system, that will only ever work for one specific type of game.

Instead, my main goal is to provide a stable, versatile and robust starting point for anyone developing games in Unity.

Writing your own character movement system, while possible, takes a lot of time and effort, with lots of pitfalls along the way. To make matters worse, character controllers, while vitally important to smooth gameplay in most games, are only ever noticed by players when things go wrong: Stuttering, falling through level geometry, coming to a complete stop at tiny obstacles...

I want to help other developers avoid these common pitfalls altogether and that's exactly why I created this package.

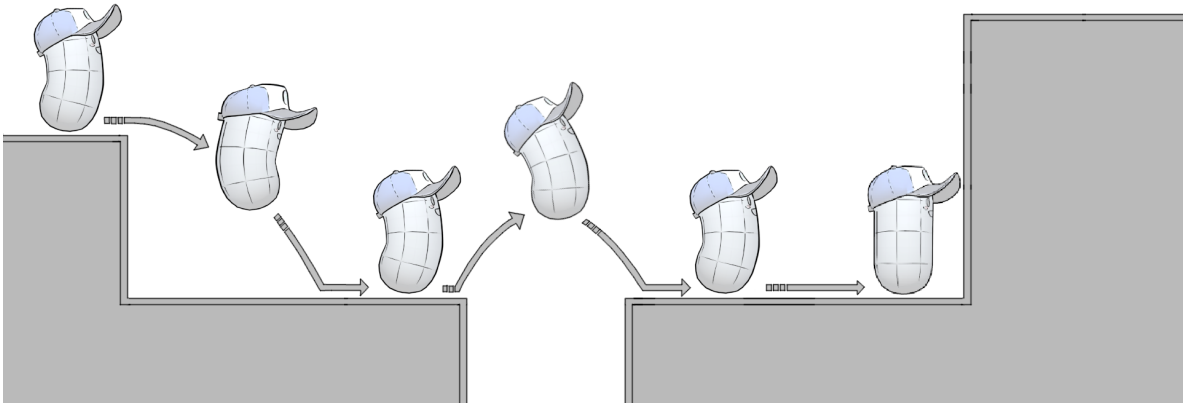


Fig. 9 *Falling, jumping and stopping at walls.*

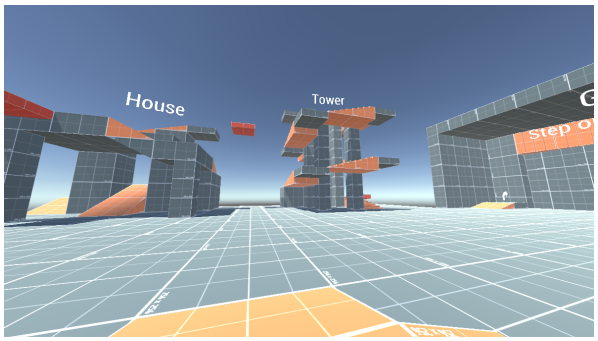
To make things as easy as possible for you, included in this package you'll find example scenes, lots of fully set up controller prefabs and fully documented scripts to help you understand what's going on behind the scenes. Even if you don't intend to write any code, you'll still be able to create controllers for almost all types of games just by modifying values in the inspector - all of the components are designed to be as versatile as possible.

2.2 What Kind of Games Can I Use This For?

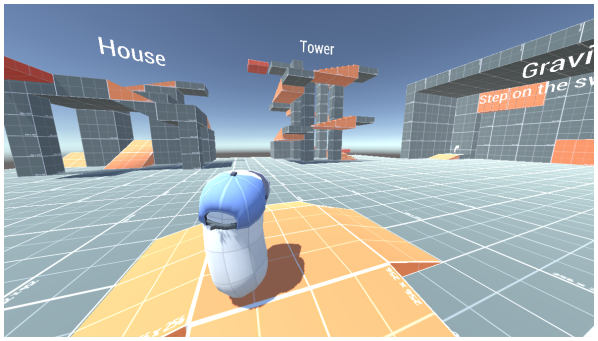
Put very simply, if your game needs characters moving around in a 3D (or 2D) environment, maybe even walking up stairs and slopes, this package is right for you!

It doesn't matter if you're planning to develop a fast-paced shooter (such as *'Doom', 'Team Fortress', 'Fortnite'*), an atmospheric adventure game (in the likes of *'Dark Souls'* or *'Spyro the Dragon'*) or even a retro 2D platformer - with just a few easy adjustments, *'Character Movement Fundamentals'* is capable of handling all of that and more.

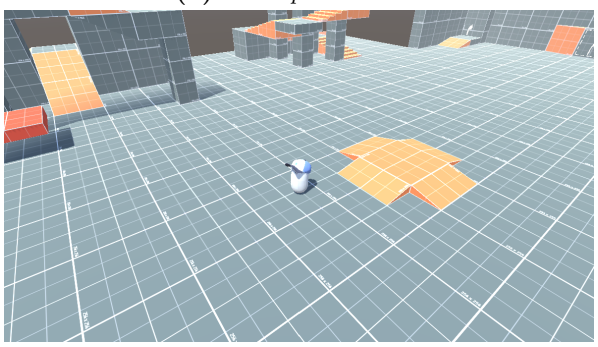
Thanks to the highly versatile camera system included in this package, experimenting with different camera perspectives is very easy - whether your game needs a first person, third person, 2D sidescroller or a top-down perspective, everything is easily configurable by changing a few values in the Unity inspector.



(a) *First person view.*



(b) *Third person view.*



(c) *A topdown game.*

Fig. 10 *Some examples of different camera perspectives and gameplay styles.*

2.3 List of Features

- Versatile, highly customizable **character controller system**
  - ◇ Based on Unity's built-in rigidbody physics.
  - ◇ Walks up and down stairs, slopes and ramps without losing ground contact.
  - ◇ Can slide down slopes that are too steep, acceptable steepness can be adjusted.
  - ◇ Adjustable movement speed, jump speed, custom gravity, air control [...]
  - ◇ Character controller can be freely rotated, even at runtime. This enables interesting gameplay possibilities - walking on ceilings and walls, on miniature planets, inside loops...
  - ◇ 3 different ground detection methods to achieve great performance on every system/device.

- All-purpose **camera system**
  - ◇ Highly customizable, can be adapted to work in a wide variety of games.
  - ◇ Built-in camera smoothing system for smooth camera rotations.
  - ◇ Adjustable camera angle limits.
  - ◇ Invertible camera axes.
  - ◇ Different sensitivity settings for gamepad and mouse input.
- **Two example scenes** to showcase the various features of the package.
- A set of **12 ready to use controller prefabs** that can be 'drag and dropped' into any scene without any further setup required.
  - ◇ A first person camera walker prefab.
  - ◇ A top-down walker prefab.
  - ◇ A 2D sidescroller prefab.
  - ◇ Three different third person camera walker prefabs.
  - ◇ All prefabs come in two versions: One version with animations and sound already set up and a blank version, for easier customizing.
- A **collection of environment assets**, which can be used for quick prototyping.
  - ◇ Basic environment building blocks (cubes, ramps, stairs, tiles), all textured and set up with materials.
  - ◇ A simple moving platform system.

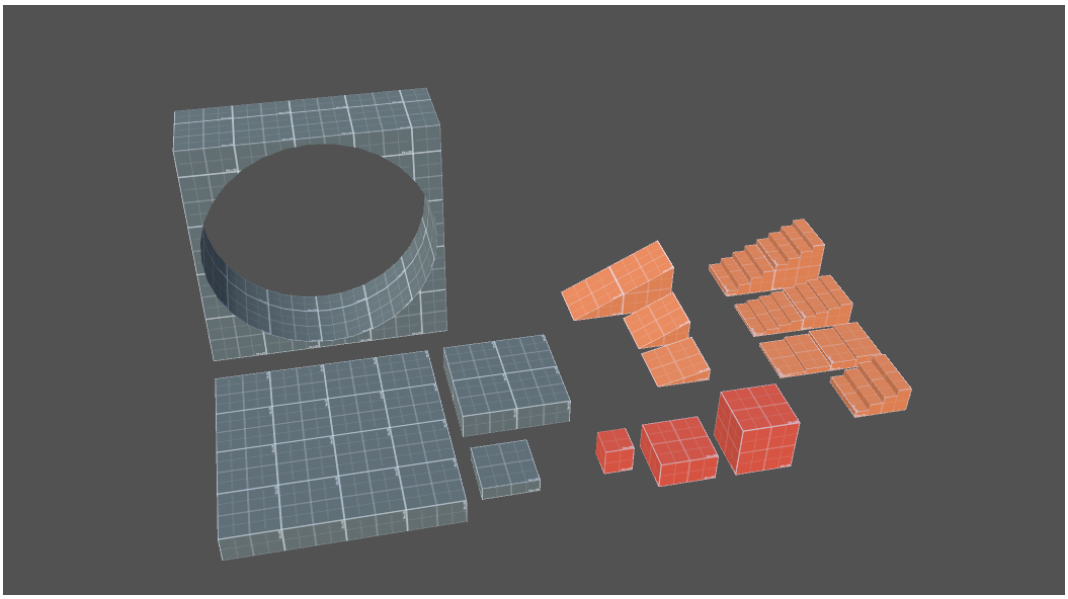


Fig. 11 *Some of the included environment building blocks...*

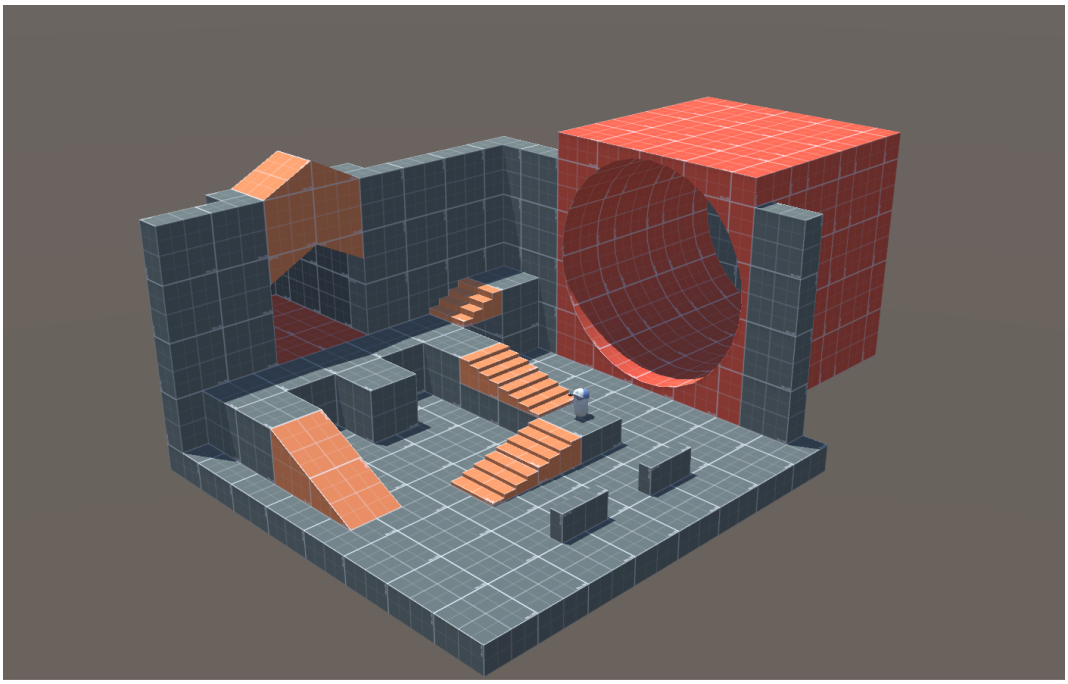


Fig. 12 *...and a simple level made using these building blocks.*

- **Rigged and animated low poly character model.**
  - ◇ Low poly character, ideal for quick prototyping.
  - ◇ Fully textured.
  - ◇ Full set of animations for walking, idling, jumping, strafing...

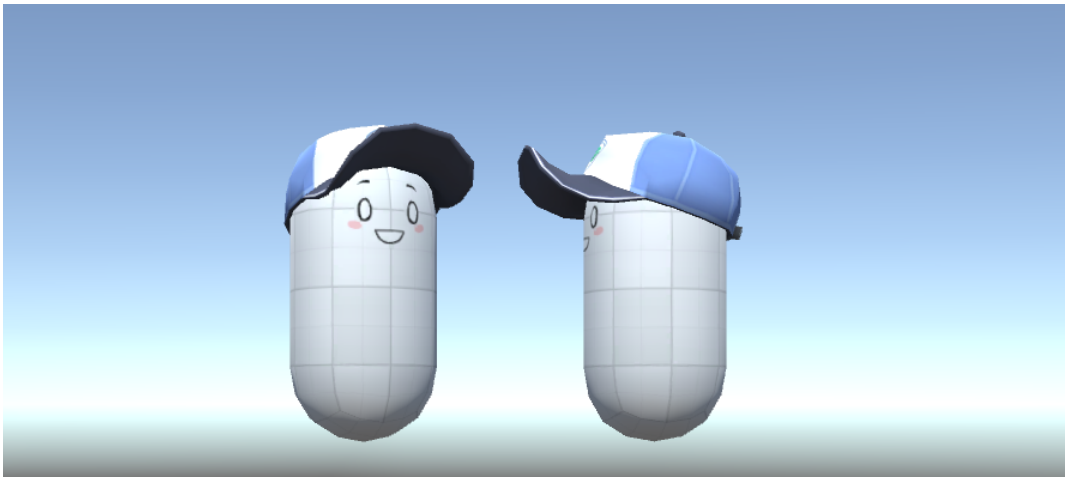
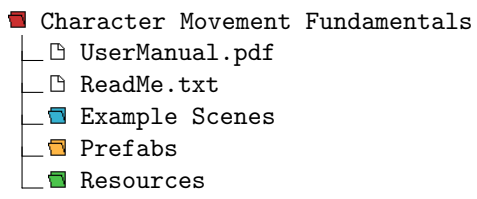


Fig. 13 *The included rigged and animated character, 'Capguy'.*

3 Package Structure

The package is structured as follows:



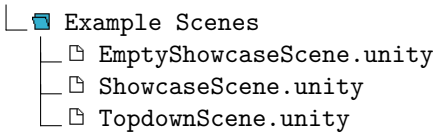
To keep everything organized, all package assets are contained contained in one root folder ('Character Movement Fundamentals'). This ensures that your project files stay separated from the package's files if the package is imported in an existing Unity project.

Inside the root folder, assets are separated by function - prefabs and example scenes are easily accessible, without having to dig around in folder hierarchies.

On the top layer, you'll also find a copy of this user manual and a readme file to keep track of any changes between different versions of this package.

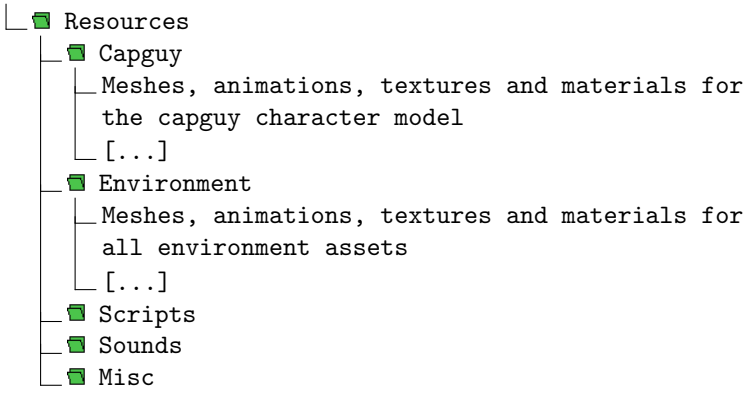
Example Scenes

This folder contains the two example scenes included in the package for demonstration purposes, as well as an "empty" scene you can use for testing and debugging.



Resources

All meshes, animations, sounds files, materials, textures and scripts can be found in this folder.

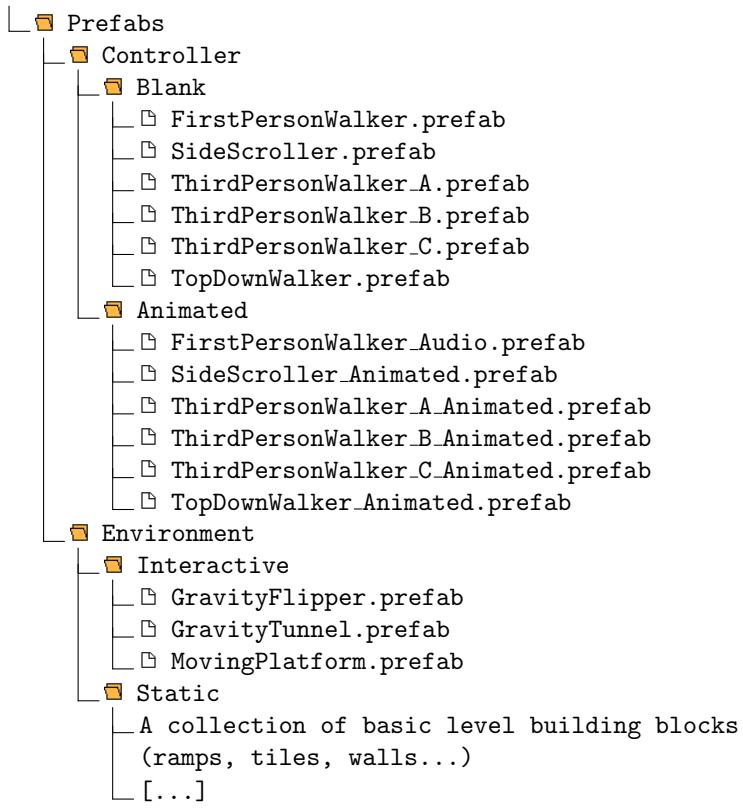


The assets are organized by their use - all files related to the environment assets are separated from the files used by the *Capguy* character (a low-poly, rigged and animated character model).

This makes it a lot easier to delete unneeded parts of the package - if you find no use for the *Capguy* character, you can safely delete its folder to reduce the overall package size.

Apart from that, 'Scripts' contains all scripts, all sound files are located in 'Sounds' and 'Misc' contains miscellaneous files like fonts, shaders and physics materials.

Prefabs
In this folder, you'll find all the prefabs included in this package.



As already mentioned, all controller prefabs come in two versions - *'blank'* and *'animated'*.

Blank controller prefabs are purposely put together using only components that are absolutely necessary for the controller to function. They are meant to be a "blank slate" or base for more advanced controllers.

Animated controllers come with basic animation (and sound) already set up and serve as an example on how to link animations and sound cues (like footsteps) with the controller's movement.

All environment prefabs are split into *'interactive'* and *'static'*. Generally speaking, all *interactive* environment assets are used in the example scenes for demonstration purposes. The *static* environment assets are composed of modular level building blocks and are used in all example scenes. They are also very useful for quick prototyping of game environments.

More detailed descriptions of each controller prefab can be found in the chapter 4.2, *'Overview Over All Included Controller Prefabs'*.

4 How to Use

4.1 Basic Principles

4.1.1 'Anatomy' of a Basic Character Controller

Mover and Controller

All Character controllers in this package follow the same basic idea: The task of moving a character around the scene is split between two components, a *'Mover'* and a *'Controller'*.

On its own, the *Mover* component won't do anything. Instead, it expects a movement velocity (and direction) from another script, which we'll refer to as a *Controller* from now on.

After having received a movement velocity, the *Mover* component will then set the rigidbody's velocity appropriately, make sure that the character maintains the correct distance from the ground and stores any useful collision information (whether the character is grounded, for example).

In the following frame, the *Controller* will use that information to determine the new movement vector, pass that vector to the *Mover* and the cycle repeats.

Or put differently, you can think of the *Mover* as the *body* of the character and the *Controller* as the *brain*. One makes the decisions, the other one carries them out.

This setup makes it possible for all characters in a game to use the same basic *Mover* component. By choosing different *Controllers*, we then can implement all kinds of different movement patterns and behaviours. We could even switch controllers during gameplay!

Object Hierarchy

In terms of gameobject hierarchy, all character controllers in this package are structured very similarly. The *Mover* and *Controller* components are added to the root gameobject of the hierarchy, any other parts (like the camera system and the character model) are added as children to that root gameobject.

```
Root gameobject (rigidbody, collider, mover, controller)
 ├─Model root
 │  └─Mesh, material, animations, [...]
 └─Camera root
    └─Camera scripts, camera, [...]
```

Usually, the character mesh/model/animations and the camera system are split into two different child hierarchies, both for practical and organizational reasons. For example, many third person games will have the camera fall slightly behind the character model, for a smoother game experience. This effect is easily achievable by using this design.

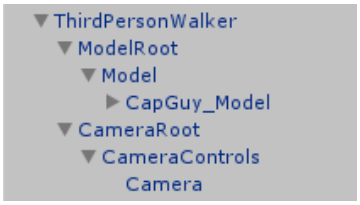


Fig. 14 *An example of a typical character controller hierarchy in the package.*

In special cases, empty gameobjects may be inserted between the different root gameobjects to prevent certain scripts from interfering with each other.

4.1.2 Three Ways to Use this Package in your Projects

Depending on your programming experience and your game's specific requirements, there's (roughly speaking) three general ways you can use this package to help you realize your projects.

a) Using one of the included prefab controllers
The prefabs included in this package should already cover most of the common types of character movement in games nowadays. Each of them is designed in a way that allows for very detailed and extensive customization, to suit almost any game's basic requirements.

For example, the first person walker prefab in this package can be easily turned into anything ranging from a fast twitch shooter (like *'Quake'*) to a slow, contemplative walking simulator, simply by adjusting some values in Unity's inspector.

Especially for quick game prototyping, game jams or even a quick test run of a level design, the included prefabs should easily cover all your bases.

In chapter 4.2, *'Overview Over All Included Controller Prefabs'*, you'll find a comprehensive overview of all the different prefabs and their individual use cases.

A detailed description of all inspector variables for all relevant components can be found in chapter 4.3, *'Component Descriptions'*.

b) Extending the included controller scripts
If your game *does* require a particularly unorthodox movement system, there's the option of extending one of the included controller scripts and overriding some of its functions to implement your own code.

This option is meant to serve as a middle way between not writing any custom code and coding your very own controller script. That way, you get to keep all the physics calculations, basic movement logic and controller states while still being able to, for example, implement a special kind of user input system.

More information on how to override specific controller functions can be found in chapter 4.4, *'How To Override Player Input Functions'*.

c) Writing a custom controller script
Finally, if you need full control over every detail of your character controller, you can also simply write a completely custom controller script yourself.

Even if this option requires some work on your part, it really isn't as difficult as it may sound at first, because the *Mover* component already takes care of most of the tricky bits related to character movement: Handling slopes and stairs, ground detection and ground normals, correctly calculating rigidbody velocities...

This option might also be a good way to go, if you've already written a controller for a different system (like Unity's built-in character controller, for example) and want to reuse your code. Adapting your existing character movement logic to fit with *Character Movement Fundamentals's Mover* component should be a fairly easy process.

Read more on how to write a custom controller script in chapter 4.5, *'Writing a Custom Controller Script'*.

4.2 Overview Over All Included Controller Prefabs

The selection of character controllers included in this package are designed to cover a wide range of gameplay types. Each one is meant to provide a good starting point for your own individual character controller, which is why you'll find all of them containing only the bare essentials needed for the character controller to function.
As mentioned in chapter 3, all character controllers come in two variants - *blank* and *animated*. Both are functionally identical, however, all *animated* variants are set up with simple animation and sound effect systems as an example on how to link character movement and animations/sounds together.
This chapter will provide a quick description of each prefab, what kind of gameplay styles they are most suited for and which special components they make use of.

All Controllers
Generally speaking, all prefabs use the following set of basic components with different settings:

- *Mover*
- *Smooth Position*
- *Smooth Rotation*

FirstPersonWalker
This is your classic first person controller - you move around using the 'WASD' keys on your keyboard (or any other set of keys of your choice) and look around by moving your mouse (or gamepad stick, for example).

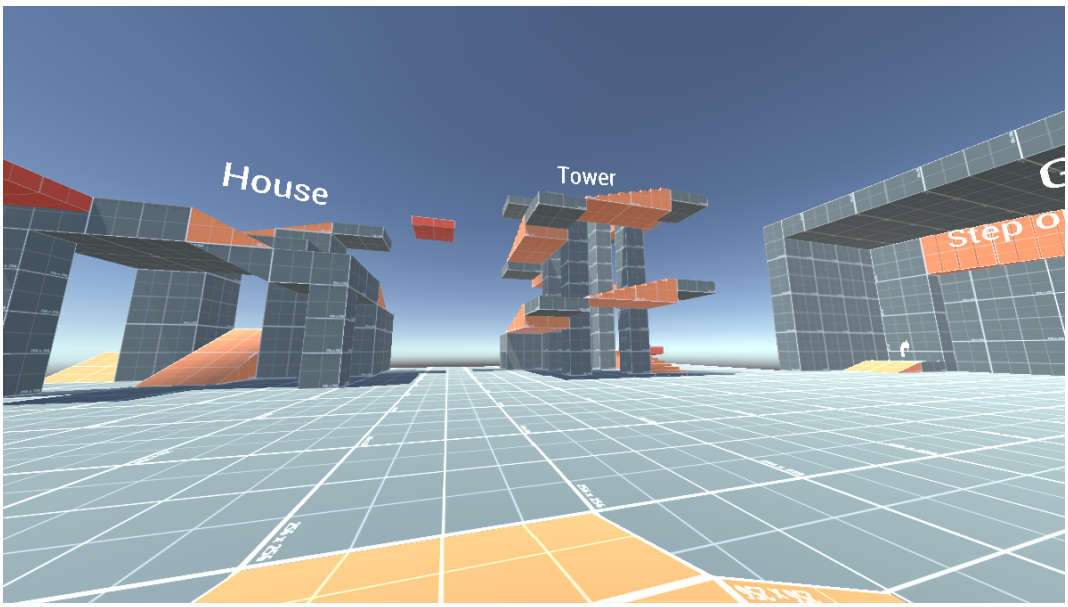


Fig. 15 *The 'FirstPersonWalker' prefab.*

As the name implies, the player camera is positioned where the character's eyes would be, giving you a first-person perspective of the game environment.

This prefab is a great starting point for any kind of first-person character controller. By changing some of its properties in the inspector (like movement speed, jump speed, air control...), you can fine-tune its movement to suit almost any type of gameplay.
It uses the following components:

- *Camera Walker Controller*
- *Camera Controller*

ThirdPersonWalker_A
This prefab serves as a basic foundation for a typical third person controller.

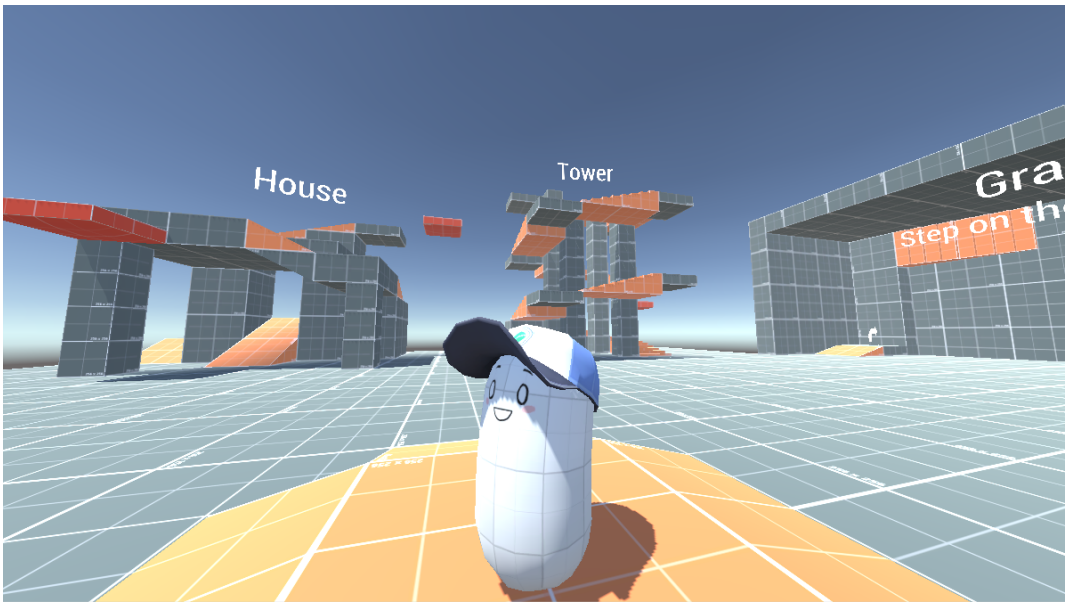


Fig. 16 *The 'ThirdPersonWalker_A' prefab.*

Movement-wise, it functions very similarly to the *FirstPersonWalker* preset, however, the player camera is positioned outside and slightly above the character's head.
To prevent the camera from going into level geometry or other objects in your scenes, all third-person prefabs also use a special component called *Camera Distance Raycaster*, which scans for any colliders blocking the camera's view.
In addition to this, the 3D model of the character is continuously rotated toward the movement direction of the controller.
It uses the following components:

- *Camera Walker Controller*
- *Camera Controller*
- *Camera Distance Raycaster*
- *Turn Toward Controller Velocity*

ThirdPersonWalker_B
This prefab is almost identical to *ThirdPersonWalker_A*, the only difference being that it uses a slightly more specialized camera controller script, which turns the camera toward the characters movement direction.
This behaviour is commonly expected by players in third person games, which are **not** primarily shooters, such as the *'Dark Souls'* or *'Spyro the Dragon'* series, for example.

- *Camera Walker Controller*
- *Third Person Camera Controller*
- *Camera Distance Raycaster*
- *Turn Toward Controller Velocity*

ThirdPersonWalker_C
This prefab is also almost identical to *ThirdPersonWalker_A*, this time however, the character model is always rotated toward the direction the camera is facing.

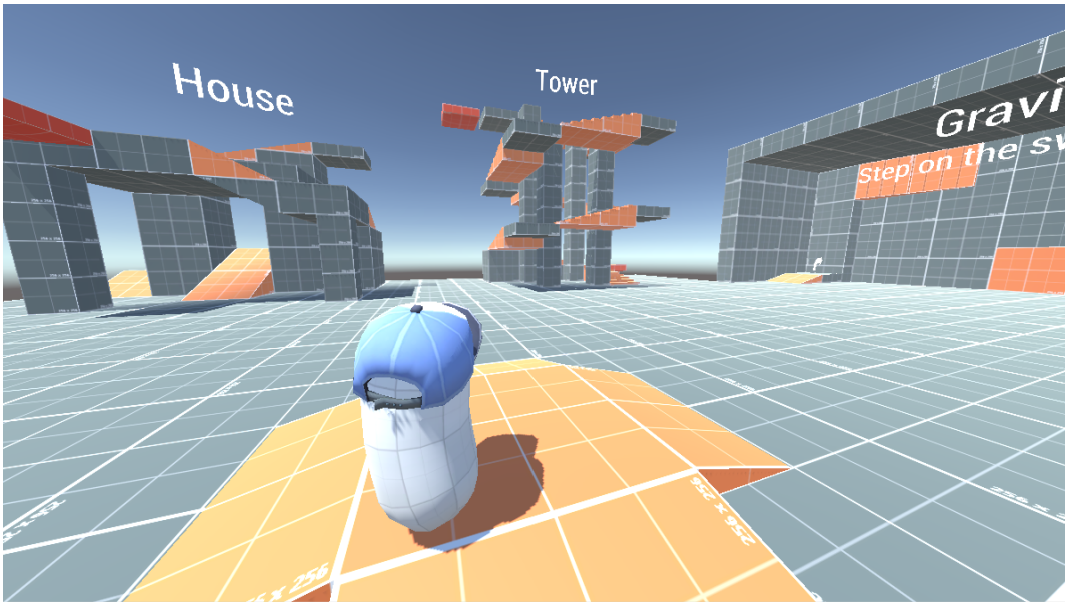


Fig. 17 *The 'ThirdPersonWalker_C' prefab.*

This makes sense for almost all third person shooters (a popular example being *'Fortnite'*), since the player character commonly aims in the same direction as the camera.

- *Camera Walker Controller*
- *Third Person Camera Controller*
- *Camera Distance Raycaster*
- *Turn Toward Camera Direction*

TopDownWalker
If you need a character controller suited for a top-down game, this prefab will meet all your basic requirements.

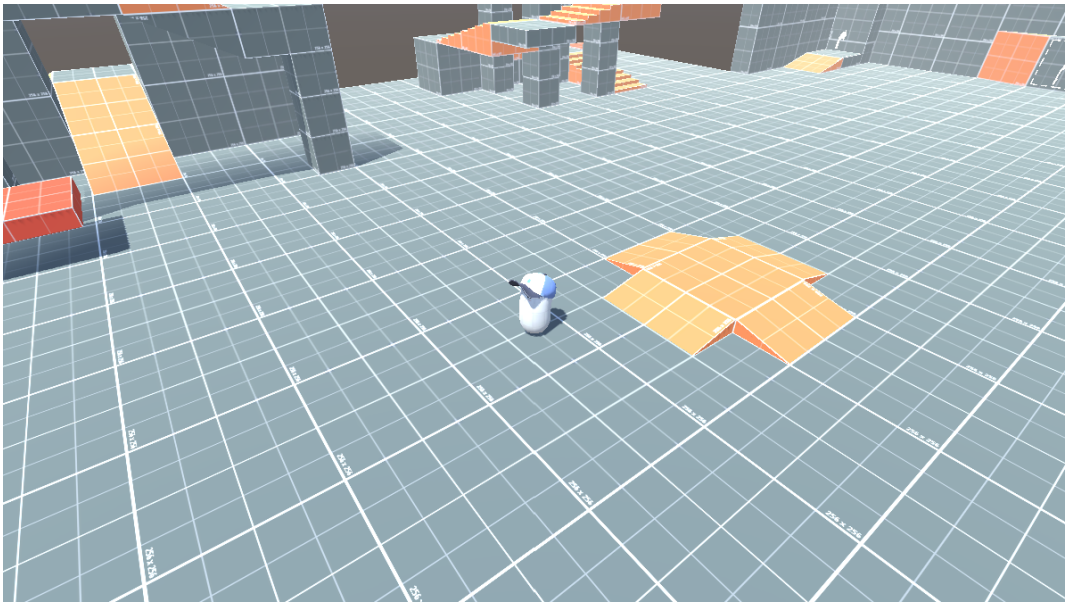


Fig. 18 *The 'TopDownWalker' prefab.*

The camera itself is offset quite far above the character and camera rotation around its x-axis (the "pitch" axis) is disabled.

- *Camera Walker Controller*
- *Camera Controller*
- *Turn Toward Controller Velocity*

SideScroller
This prefab is meant to provide a solid foundation for any 2D character controller - whether you're developing a 2D retro platformer, a sidescrolling shooter or a stealth game set in 2D, this prefab is a good starting point.

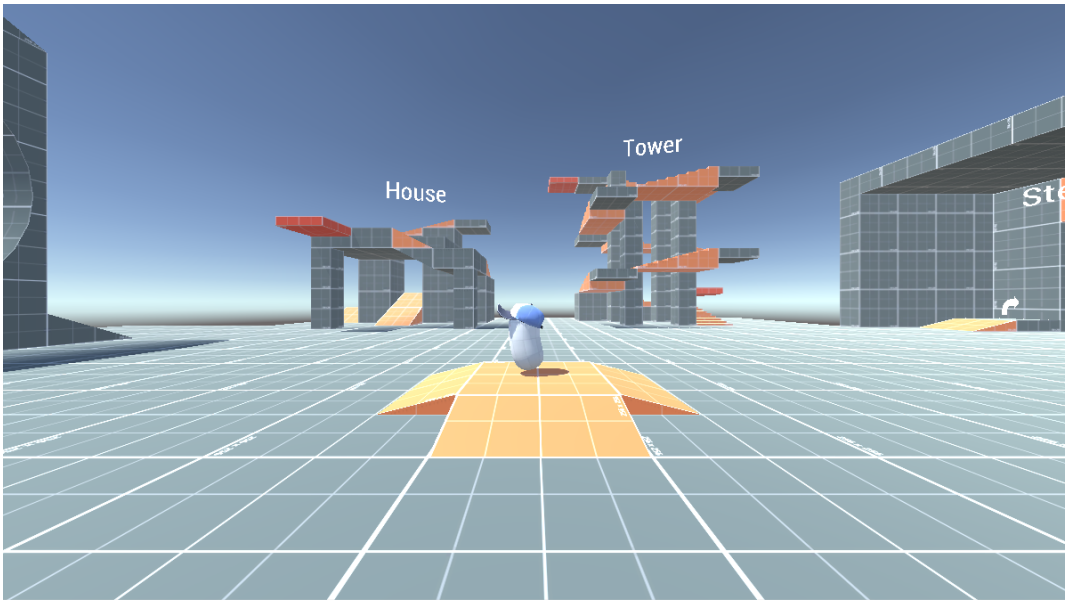


Fig. 19 *The 'SideScroller' prefab.*

In contrast to the other prefabs, character movement is locked to a 2D plane and the camera position and rotation is fixed, as is usual in most 2D games.

- *Sidescroller Controller*
- *Turn Toward Controller Velocity*

About the *animated* variants
As already mentioned, all *animated* variants of the controller prefabs are functionally identical to their *blank* counterparts.
They do, however, use a few additional components to implement animations and sounds (such as footsteps, for example).

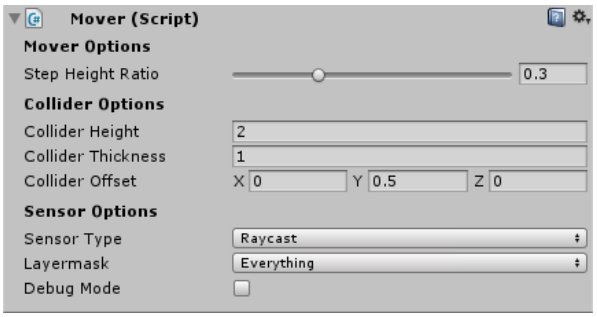
- *Animation Controller*
- *Audio Controller*

4.3 Component Descriptions

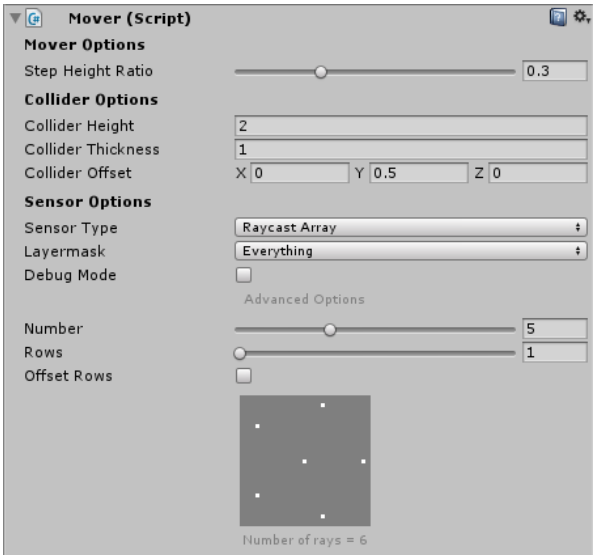
In order to use this package to its full potential, it is helpful to have a good understanding of all the components and their configurable variables available in the inspector.
This chapter will provide a thorough description of all components and their settings and how to customize them to fit your game.

4.3.1 Mover

The *Mover* component is used by all character controllers in this package to handle character movement and any interactions with the environment.



(a) *Normal view...*



(b) *...and with 'Spherecast' selected as cast type.*

Fig. 20 *The 'Mover' component in the inspector.*

It also handles resizing the character's collider and automatically updates any changes to its dimensions while the editor is running. As a result, you should use the settings provided by this script to change the shape, size and offset of the attached collider instead of directly changing settings on the collider component itself.

To optimize your game's performance, you can choose between different sensor types for ground detection.

- *Raycast* - Uses only a single raycast.
- *Spherecast* - Uses a spherecast.
- *RaycastArray* - Uses an array of raycasts. The number and distribution of raycasts can be futher specified (see figure 20b).

Different sensor types may exhibit slight differences when dealing with certain slopes and level geometry - I recommend giving each one a try to determine the best fit for your game.
The component will require a *rigidbody* and a *collider* to work. All basic Unity colliders (box, capsule, sphere) are supported, although I would recommend using the *capsule* variant, as its shape is better suited to interact with most types of level geometry.

Step Height Ratio	Acceptable step height ratio. This value ranges from '0' to '1' and is relative to the collider's height. For example, value of '0.5' means that the character will be able to walk on stairs half its height. See figure 22 for a visual example.
Collider Height	Height of the attached collider
Collider Thickness	Thickness/width of the attached collider.
Collider Offset	(Relative) position offset of the attached collider.
Sensor Type	Which type of sensor will be used for ground detection.
Layermask	Which layermask will be used when checking for ground. Use this to exclude certain level elements from being detected.
Debug Mode	If debug mode is enabled, all surface points and normals detected are displayed in the editor. Also see figure 21.

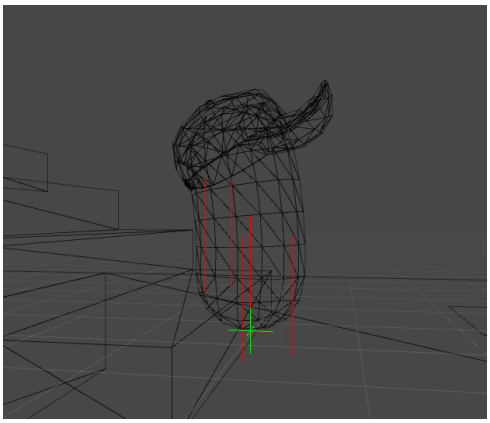


Fig. 21 *With debug mode enabled, detected surface points and normals are shown in red in the scene view. Using the wireframe view mode is recommended for better visibility. The calculated average position is shown as a green arrow.*

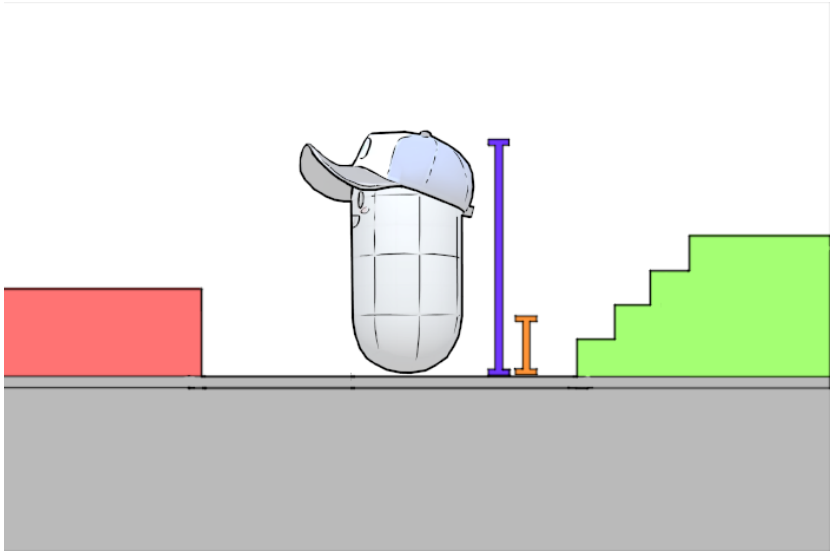


Fig. 22 *In the above image, step height ratio is set to '0.25'. All obstacles lower than the resulting height (orange), which is exactly 1/4th (= 0.25) of the characters height (shown in purple), are walkable - the blocks on the right (green) will be treated as stairs, while the block on the left (red) is considered a wall.*

4.3.2 Controllers

Basic Walker Controller
This script is used as the basis for both the *Camera Walker Controller* and *Sidescroller Controller*. It will get player input, keep track of gravity and other physical forces and calculate a movement vector, which is then passed to the *Mover* component.

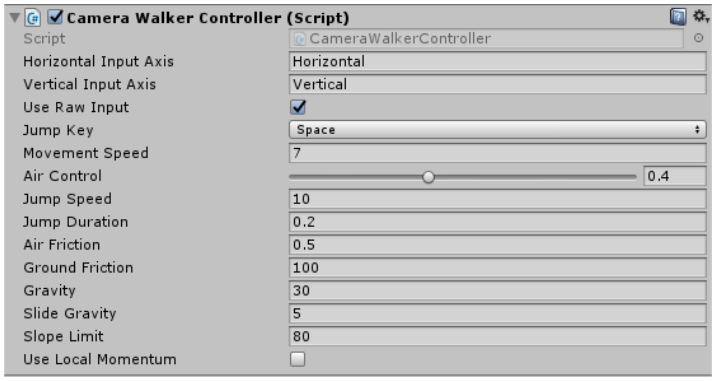


Fig. 23 *The 'Camera Walker Controller' component. Since it offers the same settings as 'Basic Walker Controller', both look identical in the inspector.*

Horizontal Input Axis	Name of input axis used for going left/right.
Vertical Input Axis	Name of input axis used for going forward/backward.
Use Raw Input	Whether to use raw input values or Unity's built-in smoothing when calculating input.
Jump Key	Key used for jumping.
Movement Speed	General movement speed of the controller.
Air Control	Amount of air control when the controller is in the air. '0' means no air control at all, '1' means complete air control. Generally speaking, more air control will result in a more responsive but less "realistic" game feel.
Jump Speed	Jump speed of the controller. Higher jump speed values will allow the character to jump higher.
Jump Duration	Maximum jump duration of the controller. If this value is set to anything above '0', the player can keep pressing the 'jump' key for a higher jump.
Air Friction	Amount of friction applied to the controller's momentum when it is in the air.
Ground Friction	Amount of friction applied to the controller's momentum when it is grounded.
Gravity	Amount of (downwards) gravity when the controller is in the air.

Slide Gravity	Amount of (downwards) gravity when the controller is sliding on a steep slope.
Slope Limit	This controls whether a surface is considered as ground or as a slope, based on its surface normal. For example, a value of '70' will allow the controller to walk on all slopes that are less steep than 70 degrees.
Use Local Momentum	If this is enabled, the controller's momentum will be calculated locally (meaning if the controller is rotated, its momentum will be rotated as well).

Camera Walker Controller
This component extends *Basic Walker Controller* and uses the camera axes provided by a 'Camera Controller' component as its movement direction.
Put simply, the character will walk where the camera is facing instead of using global transform axes. Almost all prefabs use this controller script, the only exception being the *Sidescroller Controller*.

Sidescroller Controller
This component also extends the *Basic Walker Controller* and locks (horizontal) character movement to a 2D plane defined by the character's local 'up' and 'right' transform axis.

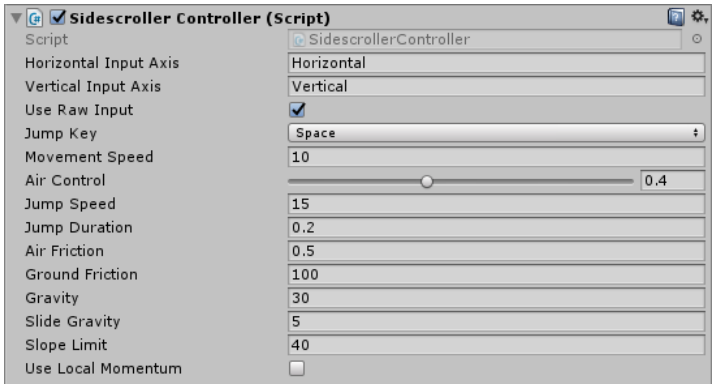


Fig. 24 *The 'Sidescroller Controller' component in the inspector.*

4.3.3 Camera Scripts

Camera Controller
This component is a general-purpose camera controller - thanks to its adaptability, you can use it in almost any game that requires a player-controlled camera.

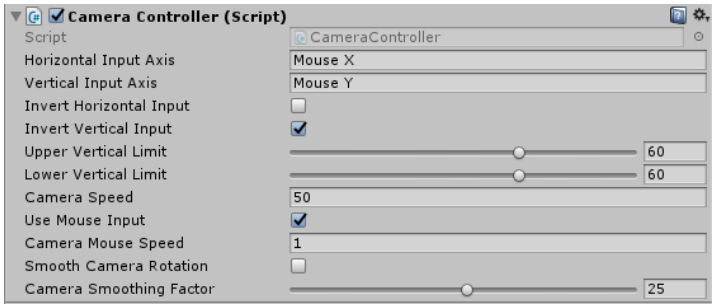


Fig. 25 *The 'Camera Controller' component in the inspector.*

It provides built-in support for inverting input axes, camera smoothing for a more "natural" game feel and options for limiting the vertical camera angle, as well as different speed settings for gamepad and mouse input.

Horizontal Input Axis	Name of input axis used for the rotation around the y-axis.
Vertical Input Axis	Name of input axis used for the rotation around the x-axis.
Invert Horizontal Input	Whether to invert horizontal input.
Invert Vertical Input	Whether to invert vertical input.
Upper Vertical Limit	Upper limit (in degrees) of the rotation around the x-axis.
Lower Vertical Limit	Lower limit (in degrees) of the rotation around the x-axis.
Camera Speed	Speed at which the camera rotates.
Use Mouse Input	Check this if you are using mouse movement as your camera input.
Camera Mouse Speed	If 'Use Mouse Input' is enabled, this camera speed will be used instead of 'Camera Speed'.
Smooth Camera Rotation	Whether to smooth player input for a smoother camera rotation.
Camera Smoothing Factor	If 'Smooth Camera Rotation' is enabled, this value will be used to smooth the camera's rotation. A value of '50' will result in no noticeable smoothing, while a value of '1' will result in very noticeable smoothing.

Third Person Camera Controller

A slightly modified version of the regular *Camera Controller*. Is is functionally identical, but adds an additional option useful for third-person games.

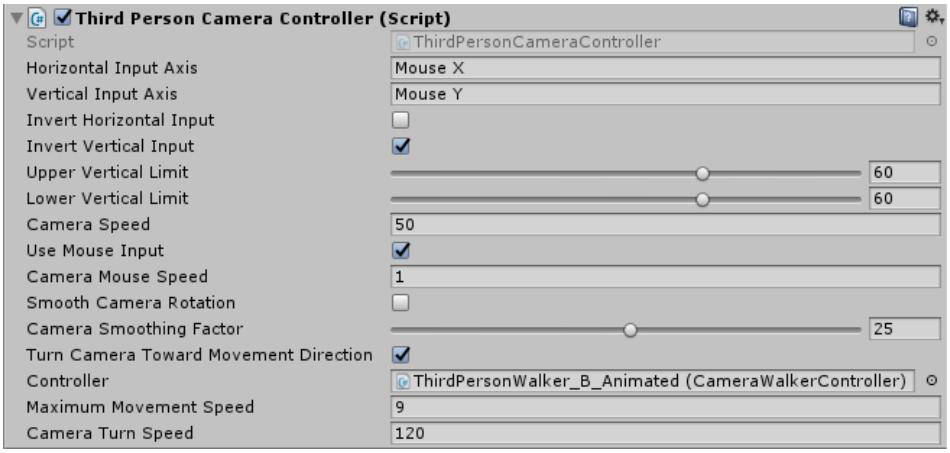


Fig. 26 *The 'Third Person Camera Controller' component in the inspector.*

If the option *'Turn Camera Toward Movement Direction'* is enabled, the camera controller will rotate toward the current movement direction of the character. This camera behaviour is commonly used in third-person games to give the player a better view of where the character is heading.
The speed of this rotation is tied to the character's movement speed, as well as the remaining angle between the camera's view and the characters movement direction.

Turn Camera Toward Movement Direction	If enabled, the camera will turn toward the character's movement direction.
Controller	Reference to a controller component. The velocity of this controller will be used to calculate the movement direction.
Maximum Movement Speed	Reference speed used when calculating the speed of the rotation toward the character's movement direction. This value should be set to the maximum movement speed achievable by the controller (i.e. the 'Movement Speed' setting).
Camera Turn Speed	Speed at which the camera turns.

Camera Distance Raycaster
This script will automatically try to detect any obstacles between camera and character and, if any obstacles are detected, will move the camera closer to the character to prevent it from clipping into level geometry.

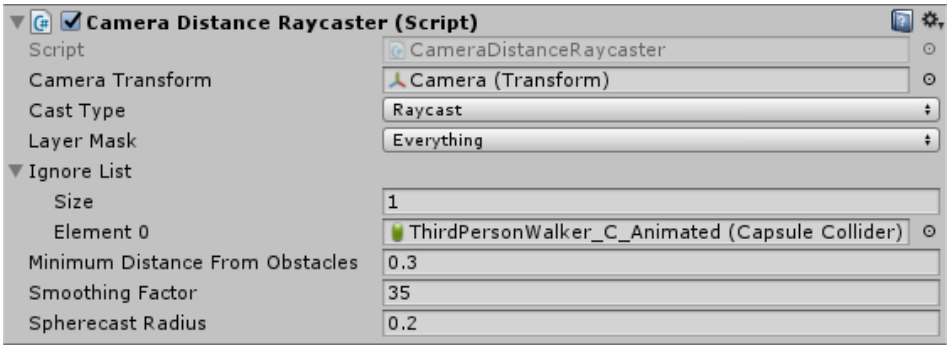


Fig. 27 *The 'Camera Distance Raycaster' component in the inspector.*

This script should be attached to the **parent transform** of the camera for it to work properly.

Camera Transform	Reference to the transform component of the camera. If this is left empty, the script will automatically choose the first child object as the camera transform.
Cast Type	Type of cast used for detecting obstacles. You can choose between either a single raycast or a spherecast.
Layer Mask	Layermask used when scanning for obstacles.
Ignore List	List of colliders to ignore while scanning. In many cases, it makes sense to add the character's main collider to this list, if you encounter any stuttering or glitches.
Minimum Distance from Obstacle	Minimum required distance from any obstacle. When an obstacle is detected, the camera is moved at least this far away from it to prevent clipping.
Smoothing Factor	This value determines how smoothly the camera position will be adjusted. A value of '50' (or greater) will result in no visible smoothing while a value of '1' (or lower) will result in extreme smoothing. I recommend trying a few different settings to find the best choice for your game. Generally speaking, '25' is a good default value for most situations.
Spherecast Radius	When *Spherecast* is chosen as cast type, this value will determine its radius.

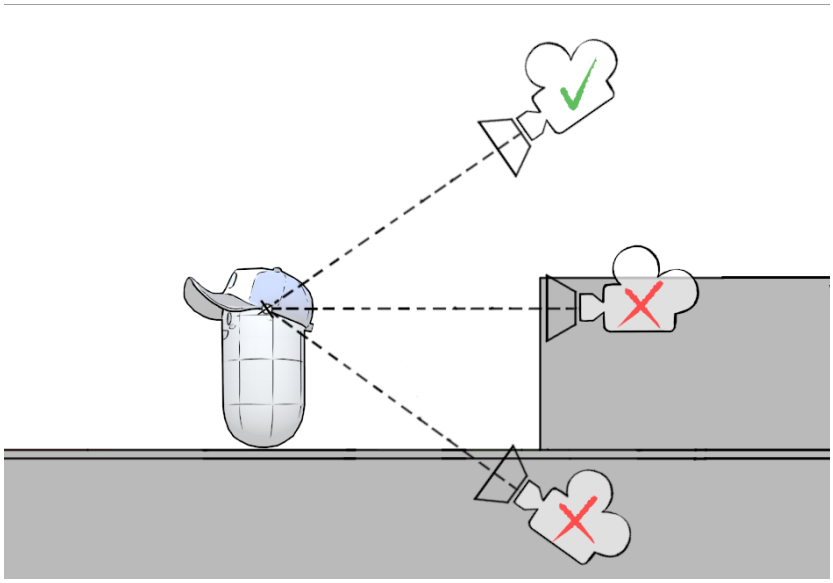


Fig. 28 *Without the 'Camera Distance Raycaster' - the camera will clip into level geometry at lower angles, causing graphical glitches.*

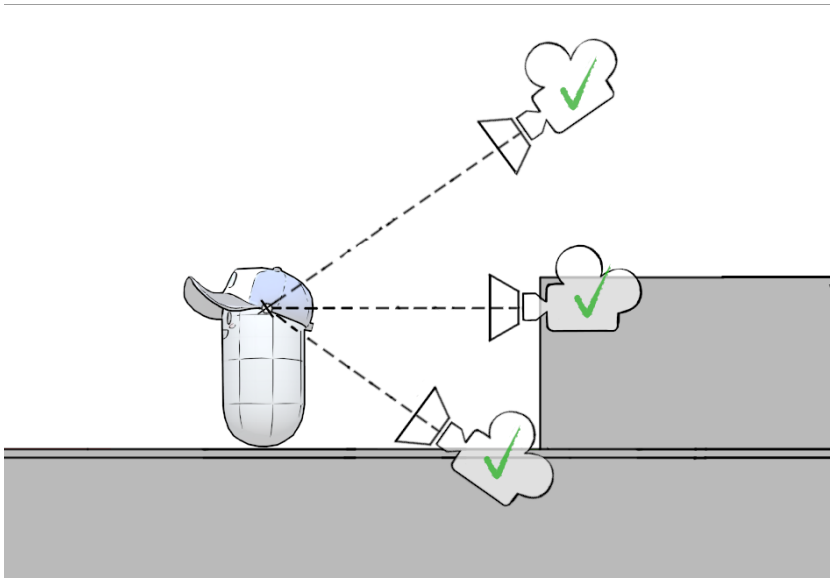


Fig. 29 *With the 'Camera Distance Raycaster'* ***enabled*** *- the camera is moved closer to the character whenever an obstacle is detected. No clipping occurs.*

4.3.4 Smoothing and Other Visual Scripts

Smooth Position
This script will smooth out the position of the gameobject it is attached to by interpolating between last frame's position and the current position.

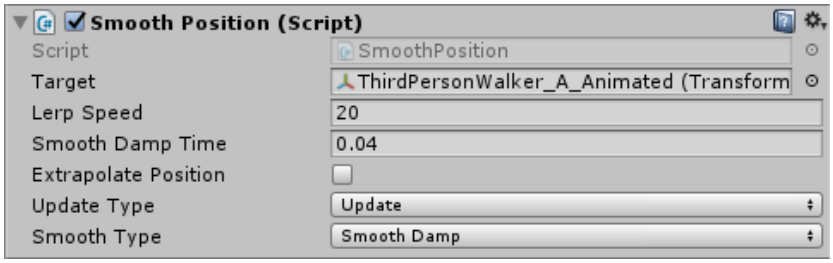


Fig. 30 *The 'Smooth Position' component in the inspector.*

It is used by all the included prefabs to smooth out some of the inconsistencies and occasional stutters, which can happen from time to time in Unity's internal physics calculations.
Alternatively, you can use it for a 'smooth camera follow' effect in third-person or 2D games.
You can choose from two different smoothing methods - Unity's built-in *'Smooth Damp'* and *'Lerp'* (linear interpolation), which will behave slightly differently. It is recommended to try both and use the option that serves your game best.
Additionally, the script can either run during Unity's *Update* or *LateUpdate* cycle, to prevent multiple smoothing scripts from interfering with each other.

Target	Target transform to lerp/smooth damp towards. When left empty, the transform's parent is automatically chosen as a target.
Lerp Speed	Speed at which the object is lerped toward its target. Used when *Lerp* is chosen as the smooth type.
Smooth Damp Time	This value determines how fast the object is smooth damped toward its target. Used when *Smooth Damp* is chosen as the smooth type.
Extrapolate Position	If this option is enabled, the target position is extrapolated to decrease some of the delay caused by the smoothing process. Note: This setting can potentially cause some 'bouncing' for very slow smoothing settings.
Update Type	Whether this script is run during *Update* or *LateUpdate*.
Smooth Type	Whether the object's position is interpolated using the *Lerp* or *Smooth Damp* method.

Smooth Rotation
This script works very similarly to *Smooth Position*, but smoothes the object's rotation instead.

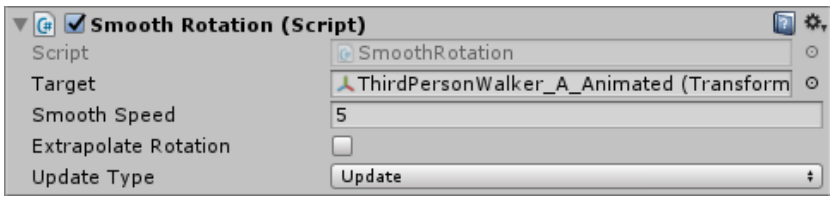


Fig. 31 *The 'Smooth Rotation' component in the inspector.*

Target	Target transform to smooth towards. When left empty, the object's parent is chosen as the target.
Smooth Speed	Speed at which the object's rotation is smoothed toward its target rotation.
Extrapolate Rotation	If this is enabled, the target rotation is extrapolated to decrease some of the delay caused by the smoothing process. Note: This setting can potentially cause some 'bouncing' for very slow smoothing settings.
Update Type	Whether this script is run during *Update* or *LateUpdate*.

Turn Toward Controller Velocity
This script rotates the gameobject it is attached to toward a target controller's movement velocity.

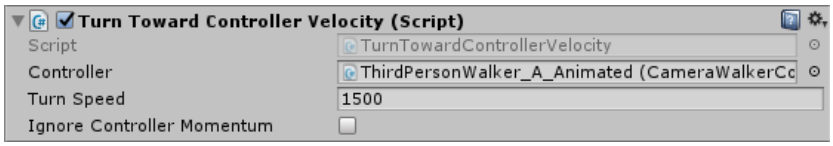


Fig. 32 *The 'Turn Toward Controller Velocity' component in the inspector.*

It is normally used to rotate a character mesh toward the movement direction of its controller component.

Controller	Reference to a controller component.
Turn Speed	Speed at which the gameobject is rotated toward the controller's movement velocity.
Ignore Controller Momentum	If enabled, only the controller's movement velocity is used to calculate the new rotation (momentum is ignored).

Turn Toward Camera Direction
This script rotates a gameobject toward the view direction of a *Camera Controller* component.

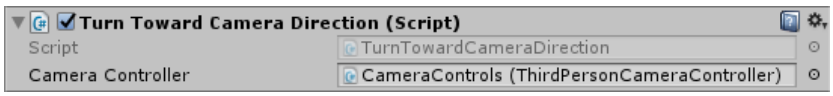


Fig. 33 *The 'Turn Toward Camera Direction' component in the inspector.*

Normally, it would be used to make a character look in the same direction as the camera, which is very common in third-person shooters, for example.

Camera Controller Reference to a camera controller component.

4.3.5 Animation and Audio

Animation Control
This script serves as a basic animation controller. It is included in this package to demonstrate how to connect animations and character movement.

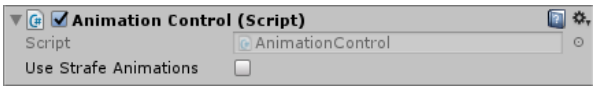


Fig. 34 *The 'Animation Control' component in the inspector.*

It requires a *Basic Walker Controller* to work and will pass necessary information (whether the character is grounded, its current speed) to an animator component.
If *'Use Strafe Animations'* is enabled, a blend tree more suited for strafing character movement will be used.

Use Strafe Animations Whether to use a blend tree better suited for strafing character movement.

Audio Control
This script provides some basic audio control features for game characters. It serves as a demonstration on how to connect animations, character movement and sound effects.

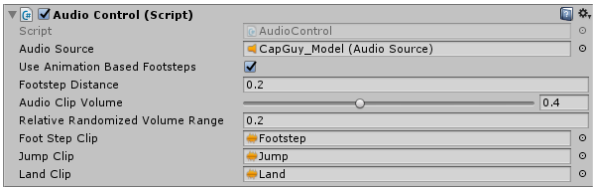


Fig. 35 *The 'Audio Control' component in the inspector.*

It requires a *Basic Walker Controller* component to work and will use the controller's velocity and its events to play footstep sounds and sound effects for jumping and landing.

In addition to that, it can trigger footstep sound either when a certain distance has been walked by the character or based on specific animation curves.
For the second option to work, you need animation curves named *'Footstep'* in your animations, to trigger a sound every time the character's animation touches the ground. Please take a look at some of the animation assets included in this package for a working example.

Audio Source	Reference to an audio source component, which will be used to play the audio clips.
Use Animation Based Footsteps	If enabled, footstep sounds will be played based on animation curves instead of based on travelled distance.
Footstep Distance	If the above option is disabled, footstep sounds will be played every time this distance is reached by the character.
Audio Clip Volume	Volume of all sound clips.
Relative Randomized Volume Range	If set to '0', all footsteps will be equally loud. If set to anything higher, footstep sounds will progressively differ in volume. This results in more natural sounding patterns of footsteps.
Foot Step Clip	Audio clip used for footsteps.
Jump Clip	Audio clip used for jumping.
Land Clip	Audio clip used for landing.

4.3.6 Environment Scripts

Moving Platform
This script serves as a basic example of a moving platform system and is included in this package mainly for demonstration purposes.

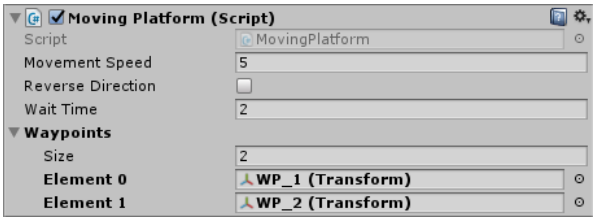


Fig. 36 *The 'Moving Platform' component in the inspector.*

It will move a gameobject along a predefined path of waypoints and move any controllers standing on it along.

Movement Speed	General platform movement speed.
Reverse Direction	If enabled, the order of waypoints is reversed.
Wait Time	How long the platform will wait every time it reaches a waypoint before continuing.
Waypoints	List of waypoints.

4.3.7 UI Scripts

Mouse Cursor Lock
This script serves provides basic mouse cursor locking functionality.

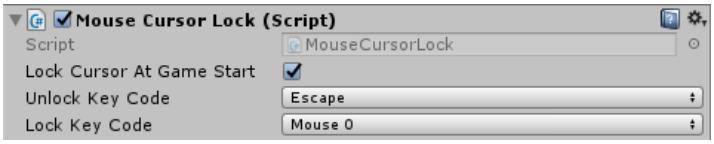


Fig. 37 *The 'Mouse Cursor Lock' component in the inspector.*

After attaching this script to any object in the scene, you can hide and show the mouse cursor by pressing assignable keys.

Lock Cursor At Game Start	Whether the mouse cursor will be locked at the start of the game.
Unlock Key Code	Key used to unlock mouse cursor.
Lock Key Code	Key used to lock mouse cursor.

4.4 How To Override Player Input Functions

4.4.1 Overriding the Basic Walker Controller

In some cases, you might need to implement a custom input system. Maybe you're using something different than Unity's built-in input handler? Maybe you're developing for a platform that requires a bit of preprocessing when it comes to user input? Maybe you are using a very unorthodox method for determining user input?

Luckily, you don't have to write a completely new controller script!

Instead, you can simply replace a relevant function in the *Basic Walker Controller* component with your own code and get the best of both worlds. All movement logic, physics calculations and handling of interactions with the environment will remain unaffected.

This is easily done by extending *Basic Walker Controller* and overriding one of its functions.

Generally speaking, there are three functions that you can override to implement your own user input code:

- *Setup()*
- *CalculateMovementDirection()*
- *IsJumpKeyPressed()*

To start off, here's a simple code example:

```
public class MyCustomController : BasicWalkerController {

    protected override Vector3 CalculateMovementDirection()
    {
      Vector3 _movementDirection = Vector3.zero;

      _movementDirection += Vector3.forward * Input.GetAxis("Mouse Y");
      _movementDirection += Vector3.right * Input.GetAxis("Mouse X");

      return _movementDirection;
    }
}
```

This custom controller extends from *Basic Walker Controller* and overrides its *CalculateMovement-Direction()* function, which is used to calculate a movement vector based on player input that is later multiplied with the character's movement speed.

Normally, this input comes from 'WASD' or arrow keys, but just to show an extreme example of a non-conventional system, the above code uses mouse input for player movement.

To change the way input is handled when it comes to jumping, things work very similarly. The *Basic Walker Controller* checks if the player has initiated a jump by calling the *IsJumpKeyPressed()* function. You can override this function like this:

```
public class MyCustomController : BasicWalkerController {

   protected override bool IsJumpKeyPressed()
   {
      //Only return true, if 'Shift' and 'Spacebar' are pressed at the same time;
      if(Input.GetKey(KeyCode.LeftShift) && Input.GetKey(KeyCode.Space))
         return true;
      else
         return false;
   }
}
```

Here, the function is overridden with code that only allows the player to jump if they press down 'Shift' and 'Spacebar' at the same time.

In case you need to initialize any variables or get any references for your custom input handling to work, you can also override the *Setup()* function like this:

```
public class MyCustomController : BasicWalkerController {

   //Reference to camera controller component;
   CameraController cameraController;

   protected override void Setup()
   {
      //Search for camera controller reference in this gameobjects' hierarchy;
      cameraController = GetComponentInChildren<CameraController>();
   }

    protected override Vector3 CalculateMovementDirection()
    {
      Vector3 _movementDirection = Vector3.zero;

      _movementDirection += cameraController.GetFacingDirection() * Input.GetAxis("Mouse Y");
      _movementDirection += cameraController.GetStrafeDirection() * Input.GetAxis("Mouse X");

      return _movementDirection;
    }
}
```

The *Setup()* function is called right after *Awake()* and is a good place to put any important declarations, initializations and function calls.

In the code example above, the *Setup* function is used to get a reference to a camera controller component in the object's hierarchy, which is later used in *CalculateMovementDirection* to calculate player movement based on the camera's view.

For more examples, please take a look at the code in the *Camera Walker Controller* and *Sidescroller Controller* components.

4.4.2 Overriding the Camera Controller

The same principles apply when overriding the *Camera Controller* component.
In order to implement custom input, you can override one of these two functions:

- *Setup()*
- *GetInput()*

GetInput() simply returns a *Vector2* variable, which stores horizontal and vertical player input. Here's a practical example of this function being overridden:

```
public class MyCustomCameraController : CameraController {

    protected override Vector2 CalculateMovementDirection()
    {
      Vector2 _input = Vector3.zero;

      _input.x = Input.GetAxis("Mouse X");
      _input.y = 0;

      return _input;
    }
}
```

This example effectively locks any vertical camera rotation, by always returning '0' as the (vertical) player input.
As for the *Setup()* function: It works identically to its counterpart in the *Basic Walker Controller*. As mentioned before, it is called right after *Awake()* and can be used for any important declarations, initializations and to store necessary references to other components:

```
public class MyCustomCameraController : CameraController {

    protected override void Setup()
    {
      //Put any initializations here;
    }
}
```

4.5 Writing a Custom Controller Script

In case your game needs some special features, which go beyond what the included controller prefabs or even overriding input functions can accomplish, there's always the option of programming a custom controller script.

Writing your very own controller script may sound difficult at first glance, but thanks to the *Mover* component, which takes care of most of the common technical difficulties of character controller programming, it's a very straightforward process. Of course, basic knowledge about programming in Unity is required, as this will require you to code.

This chapter may also be relevant for you if you've already written a character controller script in the past (maybe using Unity's built-in components) and consider re-using that code in a new project. Adapting an existing controller script to work with the *Mover* component should only require a few minor changes to your code.

4.5.1 Basic Script Structure

One very important thing to keep in mind when writing a custom controller is that **the *Mover* component runs in *FixedUpdate()***. This is because it is based on Unity's rigidbody physics system, which is run in *FixedUpdate()* to provide a stable, framerate-independent simulation. As a result, all controller scripts need to be run in *FixedUpdate()* as well.

Having mentioned that, here's the basic layout I recommend for writing a stable controller script:

```
public class MyCustomControllerScript : MonoBehaviour {

   //Reference to attached mover component;
   Mover mover;

   void Start () {
      //Get references to mover component;
      mover = GetComponent<Mover>();
   }

   void FixedUpdate () {
      //Run initial mover ground check;
      mover.CheckForGround();

      //Check whether the character is grounded;
      bool _isGrounded = mover.IsGrounded();

      Vector3 _velocity = Vector3.zero;

      //Calculate the final velocity for this frame;
      [...]

      //If the character is grounded, extend ground detection sensor range;
      mover.SetExtendSensorRange(_isGrounded);

      //Set mover velocity;
      mover.SetVelocity(_velocity);
   }
}
```

For a better understanding of all basic elements, let's go through each one separately:

```
//Reference to attached mover component;
Mover mover;

void Start () {
   //Get references to mover component;
   mover = GetComponent<Mover>();
}
```

Since the mover component will be called every physics frame, it makes sense to store a reference in *Start()* or *Awake()* to save performance.

```
void FixedUpdate () {
     //Run initial mover ground check;
     mover.CheckForGround();
```

Before doing any calculations or checks that rely on whether the character is grounded (or not), it is crucial to first run the mover's ground check function.

```
//Check whether the controller is grounded;
bool _isGrounded = mover.IsGrounded();
```

After calling the mover's ground check function, we can determine whether the character is grounded at any time. Depending on your specific character movement code, being grounded might be required for the character to jump or change the character's movement speed, among other effects.

```
Vector3 _velocity = Vector3.zero;
//Calculate the final velocity for this frame;
[...]
```

At this point in the script, you would typically put your own code to calculate the character's velocity for this physics frame. Later in this chapter, there will a basic code example on how to calculate the velocity for very basic character movement.

```
//If the controller is grounded, extend ground detection sensor range;
mover.SetExtendSensorRange(_isGrounded);
```

To allow the character to move **down** terrain, slopes and stairs, is is necessary for the mover's ground check range to be extended when the character is grounded. This is meant to better approximate what walking down stairs or slopes *feels* like: A **continuous motion** instead of constantly losing and regaining ground contact.

```
//Set mover velocity;
mover.SetVelocity(_velocity);
```

Finally, the movement velocity for this frame is passed to the mover component. One important thing to keep in mind here is that the velocity **must not** be multiplied with *Time.deltaTime*.

Kinematic controllers, such as Unity's built-in character controller, basically expect an instruction like *"Move in this direction by this distance, right now!"*, while **rigidbody-based controllers**, like the controllers in this package, need to be told something like *"This is your movement velocity, use it later to calculate your new position when the physics simulation is run!"*.

I highly recommend sticking to this script outline, unless you plan on heavily modifying the *Mover* component itself (which would go beyond the scope of this user manual).

4.5.2 A Basic Controller Code Example

Using the basic code outline discussed in the previous chapter, all that's left for a functional controller script is to calculate the character's velocity.
To keep this example practical, let's assume that our character needs to be able to walk around at a changeable speed, as well as jump and fall according to a custom gravity.
The first step should be to add some variables to the basic script outline:

```
float currentVerticalSpeed = 0f;
bool isGrounded;

public float movementSpeed = 7f;
public float jumpSpeed = 10f;
public float gravity = 10f;
```

In order to let the character jump and fall in a believable way, we need a variable to store its current vertical speed, aptly named *currentVerticalSpeed*.
This variable's value will change when the character jumps (by adding speed based on *jumpSpeed*) and when it falls (by subtracting speed based on *gravity*).
In addition to that, we'll also add *movementSpeed*, so we can change the character's movement speed if needed and a simple boolean variable to track whether it is grounded (*isGrounded*).

```
//Run initial mover ground check;
mover.CheckForGround();

//Check whether the character is grounded and store result;
isGrounded = mover.IsGrounded();
```

Next, after having run the mover's ground check function, we store the result in our *isGrounded* variable.

```
Vector3 _velocity = Vector3.zero;

//Add player movement to velocity;
_velocity += Vector3.forward * Input.GetAxis("Vertical") * movementSpeed;
_velocity += Vector3.right * Input.GetAxis("Horizontal") * movementSpeed;
```

On to calculating velocity: We'll start by adding the characters movement for this frame, which consists of the player's input and the character's movement speed multiplied with two global direction vectors.
Next up is the character's custom gravity.

```
//Handle gravity;
if(!isGrounded)
{
   currentVerticalSpeed -= gravity * Time.deltaTime;
   _velocity += Vector3.up * currentVerticalSpeed;
}
else
{
   if(currentVerticalSpeed <= 0f)
      currentVerticalSpeed = 0f;
}
```

Essentially, the above code checks if the character is grounded and if not, subtracts from its current vertical speed, based on our variable *gravity*.
However, if the character **is** grounded, we'll get rid of any negative vertical speed - after all, we don't want any forces pressing the character down, if it is already touching the ground.

```
//Handle jumping;
if(isGrounded && Input.GetKey(KeyCode.Space))
{
   currentVerticalSpeed = jumpSpeed;
   isGrounded = false;
}
```

Right after that, we will implement jumping, by simply checking if *a)* the character is grounded as well as *b)* the player is pressing down the *spacebar* on their keyboard.
If both conditions are met, we'll replace the current vertical speed with the characters jump speed, effectively adding a lot of upwards speed in a split second, thus approximating a jump.
Furthermore, we also need set *isGrounded* to *false*, since our character clearly can't be grounded anymore, after having just initiated a jump.

```
//Add vertical velocity;
_velocity += Vector3.up * currentVerticalSpeed;

mover.SetExtendSensorRange(isGrounded);
mover.SetVelocity(_velocity);
```

The resulting vertical speed is then added to the final velocity by multiplying it with the global "up" vector in the scene.
After that, all that is left to do is to extend the mover's ground check range if the character is grounded and to finish our script, pass the resulting final velocity to the mover component, which will take it from here.
Finally, here is the finished script in its entirety:

```
public class MyCustomControllerScript : MonoBehaviour {

   private Mover mover;

   float currentVerticalSpeed = 0f;
   bool isGrounded;

   public float movementSpeed = 7f;
   public float jumpSpeed = 10f;
   public float gravity = 10f;

   void Start () {
      //Get references;
      mover = GetComponent<Mover>();
   }

   void FixedUpdate () {
      //Run initial mover ground check;
      mover.CheckForGround();

      //Check whether the character is grounded and store result;
      isGrounded = mover.IsGrounded();

      Vector3 _velocity = Vector3.zero;
```

```
        //Add player movement to velocity;
        _velocity += Vector3.forward * Input.GetAxis("Vertical") * movementSpeed;
        _velocity += Vector3.right * Input.GetAxis("Horizontal") * movementSpeed;

        //Handle gravity;
        if(!isGrounded)
        {
            currentVerticalSpeed -= gravity * Time.deltaTime;
        }
        else
        {
            if(currentVerticalSpeed <= 0f)
                currentVerticalSpeed = 0f;
        }

        //Handle jumping;
        if(isGrounded && Input.GetKey(KeyCode.Space))
        {
            currentVerticalSpeed = jumpSpeed;
            isGrounded = false;
        }

        //Add vertical velocity;
        _velocity += Vector3.up * currentVerticalSpeed;

        mover.SetExtendSensorRange(isGrounded);
        mover.SetVelocity(_velocity);
    }
}
```

Please refer to the *Basic Walker Controller* and the scripts which extend it, for examples of more advanced controller scripts. All code is fully documented and, even though it is considerably more complex, still follows the same basic code layout described in this chapter.

4.5.3 Moving a Controller in Relation to the Camera's View

Instead of using global transform axes (like *Vector3.up*), as we did in the example controller script, it makes more sense for games that use a player-controlled camera for the character to move **in relation to the camera's view axes**.
This behaviour is easily implemented using these three functions provided by the *Camera Controller* component.

- *GetFacingDirection()*
- *GetStrafeDirection()*
- *GetAimingDirection()*

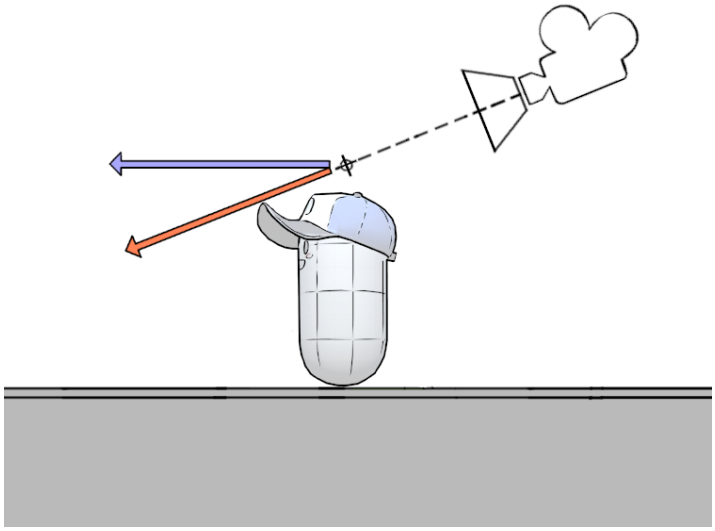


Fig. 38 *The vectors returned by 'GetAimingDirection'(orange) and 'GetFacingDirection' (blue).*

As seen in the diagram above, each function will return a different vector representing either the direction the camera is facing in, the corresponding 'strafing' direction or the direction the camera is pointing (or aiming) at.
For character movement, you should use the *facing* direction to move the character forward and the *strafe* direction to move it sidewards.
The *aiming* direction can be used to fire projectiles or weapons in shooter games, to raycast for specific game objects or to determine where the player is currently looking at.
To use these functions in your custom controller code, simply replace the following lines:

```
Vector3 _velocity = Vector3.zero;

//Add player movement to velocity;
_velocity += Vector3.forward * Input.GetAxis("Vertical") * movementSpeed;
_velocity += Vector3.right * Input.GetAxis("Horizontal") * movementSpeed;
```

With something similar to this:

```
    Vector3 _velocity = Vector3.zero;

   //Add camera direction vectors;
   _velocity += cameraController.GetFacingDirection() * Input.GetAxis("Vertical") * movementSpeed;
   _velocity += cameraController.GetStrafeDirection() * Input.GetAxis("Horizontal") * movementSpeed;
```

Naturally, your character's hierarchy needs to contain a *Camera Controller* component and you would have to get a reference to this component first by using the *Start* or *Awake* functions in your custom script.

4.6 Writing External Scripts

4.6.1 Adding Forces to a Controller

Since the *Mover* component is directly controlling (and overriding) the rigidbody's velocity, using the default physics functions for adding forces like *'AddForce()'* will have no effect.
Instead, the *Basic Walker Controller* component provides its own function to add forces:

```
public void AddMomentum (Vector3 _momentum)
```

Calling this function from an external script allows you to directly add to the controller's momentum without causing any problems with its internal movement logic.
After that, the newly added momentum will be affected by the controller's friction and gravity settings.

```
void LaunchUpwards()
{
   //Add upwards momentum to controller;
   controller.AddMomentum(Vector3.up * 100f);
}
```

For example, the above code would instantly launch the controller directly upwards, which could be used for a "jump pad" effect.

4.6.2 Rotating the Camera toward a Target Direction or an Object in the Scene

In many games, making the camera look at a specific object in the scene (or in a specific direction) is a simple but efficient way to direct the player's attention.
Other times, you might even want to give the player direct control over this camera behaviour, so they can "lock on" to enemies in third-person games (as seen in games like *'Dark Souls'*).
To make implementing this kind of camera movement easier, the *Camera Controller* component offers the following two functions:

```
public void RotateTowardPosition(Vector3 _position, float _lookSpeed)
public void RotateTowardDirection(Vector3 _direction, float _lookSpeed)
```

Using them in your code only requires you call one of these functions from an external script and pass either the position of an object in the scene or a direction vector you calculated as well as a speed value (to control how much the camera will turn in that one frame).

```
void Update()
{
   if(Input.GetKey(Keycode.Q)
   {
      //Rotate camera toward 'targetObject';
      cameraController.RotateTowardPosition(targetObject.position, 45f);
   }
}
```

In the code example above, the camera is rotated toward a target object as long as the player is holding down the 'Q' key on their keyboard.
However, it is important to keep in mind that the function has to be called continuously over many frames for the camera to eventually look in the intended direction - as a result, it is best used in either *Update()* or as part of a coroutine.

4.7 How to Connect Animations and Audio to Character Movement

The addition of animations and sounds to a character will make it feel more believable as well as add more feedback for the player to react to. Some practical examples of this include:

- Triggering footstep sound clips whenever a character reaches a certain point in its walking animation.
- Passing a characters current movement speed to a blend tree for believable animated character movement.
- Adding a subtle head bobbing motion to your player camera, when the character is moving.
- Activating a screen shake animation when the player character lands on the ground after a high jump.

In order to make implementing all of this easier, *Basic Walker Controller* features **getter functions** and **events** to help you access all the necessary information you might want to pass to your animator components and audio scripts.
Already included in the package are two scripts, *Animation Control* and *Audio Control*, which are used by all the *animated* prefab character controllers as a demonstration on how to realize some of the examples mentioned above.
Both scripts are fully documented and I recommend referring to them when programming more advanced animation and audio control scripts.

Getter Functions

GetVelocity()	Returns the current velocity of the character as a *Vector3*
GetMovementVelocity()	Returns only the current movement velocity (without momentum) of the character as a *Vector3*
GetMomentum()	Returns the current momentum of the character as a *Vector3*
IsGrounded()	Returns *true*, if the character is currently grounded.

Here's a basic code example of these values being retrieved and passed to an animator component:

```
public class MyAnimationControl : MonoBehaviour {

   BasicWalkerController controller;
   Animator animator;

   void Start()
   {
      controller = GetComponent<BasicWalkerController>();
      animator = GetComponentInChildren<Animator>();
   }

   void Update () {
      //Get controller velocity;
      Vector3 _velocity = controller.GetVelocity();

      //Get controller grounded status;
      bool _isGrounded = controller.IsGrounded();
```

```
      //Pass values to animator component;
      animator.SetBool("IsGrounded", _isGrounded);
      animator.SetFloat("Speed", _velocity.magnitude);
   }
}
```

Events

OnJump() This event is called when the character initiates a jump.

OnLand() This event is called when the character lands after being in the air.

In both cases, the current velocity of the character (at the time the event was called) is provided as a *Vector3*. You can use this vector to (for example) determine the intensity of the animation which accompanies the event or to calculate a fitting audio volume for a sound cue.
You can use these events in your own code by connecting your own functions to them, like this:

```
public class MyControllerEventHandler : MonoBehaviour {

   BasicWalkerController controller;
   Animator animator;
   AudioSource audioSource;

   void Start()
   {
      controller = GetComponent<BasicWalkerController>();
      animator = GetComponentInChildren<Animator>();
      audioSource = GetComponentInChildren<AudioSource>();

      //Connect events to controller events;
      controller.OnLand += OnLand;
      controller.OnJump += OnJump;
   }

   void OnLand(Vector3 _v)
   {
      //Set animation trigger;
      animator.SetTrigger("OnLand");

      //Only play audio clip if velocity magnitude surpasses a certain threshold;
      if(_v.magnitude > 10f)
         audioSource.Play();

   }

   void OnJump(Vector3 _v)
   {
      //React to event by playing audio clip, animations, [...]
      animator.SetTrigger("OnJump");
   }
}
```