2D Platform Controller v1.7.0

JNA Mobile

Get support on the Unity Forums by contacting JohnnyA

Get support via email by contacting support@jnamobile.com

Character Basics 2
- Quick Start 2
- Changing the model 2
- Updating the Raycasts 2
- Number of Raycast Colliders 3
- Layers 3
- Auto Settings 3
- Direction Checkers (NEW IN 1.5) 4
- Ledge Hanging (NEW IN 1.5) 4
- Wall Slide (NEW IN 1.5) 5
- Crouch and Crouch Slide (NEW IN 1.6) 6
- Animations 8
- Ladders (UPDATED IN 1.7) 9
- Ropes 10
- Triggers and Colliders 10

Input 11
- Touch Input 11

Animation 12
- One Frame States 12
- Working in UnityScript (Unity JavaScript) 12

Extra Features 14
- Enemies 14
- Damage 15
- Collectables 15

Samples Requiring Third Party Assets 16
- 2D Toolkit 16

Character Basics

Quick Start

To familiarize your self with the controller use the Sample Scenes. The **HeroSample** is the newest sample and includes an animated 3D character with ledge hanging and rope climbing.

For your own scenes it is recommended that you start with the prefab **BasicCharacter**. Drag this on to your scene. This square character is controlled with arrow keys and space bar and will interact with unity colliders.

Alternatively for 3D models or for testing ledge climbing you can use the **HeroCharacter**.

If you are having issues ensure the z-axis is aligned for all the objects (for example all of them have z = 0).

The controller ONLY works in an X-Y plane.

Changing the model

1) Remove the mesh renderer from the square controller.
2) Add your model or 2D sprite as a child of the controller.
3) Do not rotate the controller object. If you need to adjust rotation rotate the child object.

Updating the Raycasts

Use the editor gizmos to change offset and distance parameters for your raycasts. Use the **Collider Editor Options** to show only the handles that you need to use. Alternatively enter values in the Inspector.

The green raycasts are downwards and should be used as feet colliders, they push your character up from the ground. They are also used to detect slopes and ladders. Feet colliders should be the same length and at the same *y* offset as each other, use the **Align Feet** button to ensure this is the case.

The purple raycasts are upwards and should be used as head colliders. They push your character downwards from obstacles above them.

The red and yellow colliders are left and right colliders. They push your character away from obstacles to the left or right.

Number of Raycast Colliders

For a simple controller you can use 2 feet, 2 head, and 4 side raycasts (2 left, 2 right).

A **climbing** controller will need 3 feet.

A **sloped** controller will need 3 feet colliders, for better results particularly with steep platforms use 5 feet colliders.

The gap between your **side** colliders, should be smaller than the thinnest platform colliders. If you want complex shapes or thin platforms you will likely need to add additional **side** colliders.

For mobiles you may need to restrict the total number of colliders. Note that this is usually only a problem if you have multiple characters, even low-end mobiles can support 12 or more raycasts without issue.

Layers

There are three layers used by the controller:

> The **background** layer is for normal platforms.
>
> The **pass through** layer lets characters walk and jump through the platform, but they can still stand on it.
>
> The **climbable** layer is used for ladders, vines, and other climbable objects. See the notes on ladders for setting up climbable objects.

Other layers are ignored.

Auto Settings

> **Climbing.AutoStick** – Causes the character to automatically grab on to ladders, ropes and other climbables. If set to false the player needs to hold up or down to latch on to ladders and ladders, ropes and other climbables.
>
> **LedgeHanging.AutoGrab**– Causes the player to automatically grab on to a ledge if near it. If set to false the player must hold towards the ledge to grab.

Direction Checkers (NEW IN 1.5)

The direction checker is responsible for determining which way your character is facing, and what to do with colliders when his direction changes. The in-built behaviour and the default **DirectionChecker** face the same direction as the character is moving (velocity.x). Instead you might want to create your own which faces the direction the player is holding, or one that always faces the enemy.

Direction Checkers are also useful if you have a character that is not symmetrical. When your character changes direction you can invoke the characters SwitchColliders() method so that his left side collider become his right side, and vice-versa. Alternatively you can override and write your own SwitchColliders method.

Ledge Hanging (NEW IN 1.5)

Ledge Hanging is a new feature where a character can grab on to a ledge as they fall past it and then climb it.

Auto Grab will make sure character automatically start a ledge grab if they are within the **Auto Grab Distance.** If its off you need to press towards the ledge.

The various offset values express the relationship between your ledge hang animation roots and the character controllers transform.

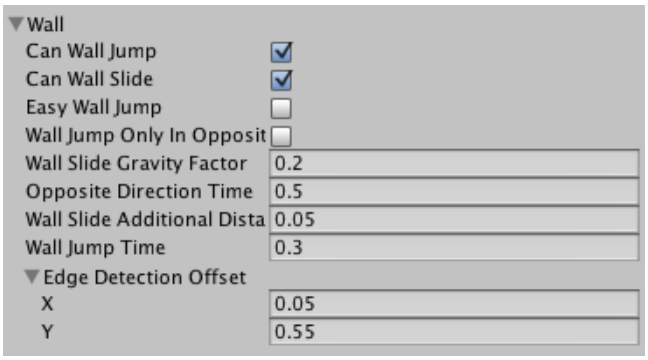
The **Ledge Drop Time** occurs when you drop off from the ledge (by pressing away from the wall or down). During this time you will not collide with your feet. This is useful for tiled or complex geometry you can usually set it to zero if you don't use tiles.

Animating a ledge hang and climb can be quite tricky consult the **HeroSample** or contact support if you need help!

Wall Slide (NEW IN 1.5)

Users upgrading from 1.5 or below please note that some of the jump details are now in the new wall details settings. Ensure you copy settings from the old to the new place.

Wall slide allows a character to cling to the wall to reduce their fall speed. Change the **Wall Slide Gravity Factor** from 1 (no effect) through to 0 (cling to wall).

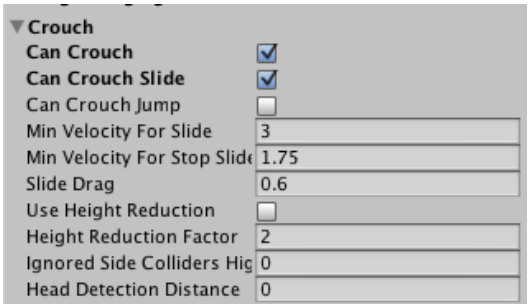


To start the slide hold towards the wall. To jump from the wall press jump and the opposite direction at the same time. If you have **Easy Wall Jump** on then you don't need to press the opposite direction key.

The **Edge Detection Offset,** ensures you don't start clinging to a wall until you are far enough down the wall. Change the y value up or down to suit your wall cling animation.

Crouch and Crouch Slide (NEW IN 1.6)

Crouch allows a character to duck, and optionally automatically reduces their height. Crouch slide allows a character to slide along whilst crouching. Crouch is triggered when the y direction is set to -1.



The velocity settings control the speed required to start the slide and the speed where a slide will stop.

The **drag** controls the rate at which your slide slows down.

Use Height Reduction will automatically make the character smaller when they duck. The height colliders will be scaled by the provided factor and moved downwards by half the **Heigh Reduction Factor**. Any side colliders above the **Ignore Side Colliders Height** will not be used when calculating if the character has hit a platform.

The **HeadDetectionDistance** stops you un-crouching whilst something is above your head.

To get a better understanding the recommendation is to put the game view and scene view side by side and test the effect of the automatic height reduction on the characters colliders.

If you don't use automatic height reduction you have two options:

> In 3D you can attach your colliders to the characters head or shoulder bones hence syncing them with the animation. Alternatively here you could add a bone just to control the ducking colliders (this is a good way to avoid the extra movement associated with using key bones).
>
> In 2D you can create your own transform and listen for the character state changes CROUCHING and CROUCH_SLIDING. When these states are sent you can automatically adjust this transform, either immediately or smoothly using (for example) iTween or a Coroutine. Don't forget to adjust them back when the state changes to something else (use previous state to check for this).

Platform Behaviour

Users upgrading from 1.1 or below please note that the signatures for some of the Platform methods have changed. You will need to update any custom platform code.

To add special behaviour to your platforms extend the Platform class.

The most interesting method is DoAction:

```
/// <summary>
/// This is called when a platform is hit. Override to
/// implement platform behaviour.
/// </summary>
/// <param name='collider'>
/// The collider that did the hitting.
/// </param>
/// <param name='character'>
/// The character that did the hitting.
/// </param>
virtual public void DoAction(RaycastCollider collider,
                             RaycastCharacterController character)
```

Use this method like a trigger. For example consider the following:

```
public class DissappearingPlatform : Platform {
    public BoxCollider myCollider;
    public MeshRenderer myRenderer;
    override public void DoAction(RaycastCollider collider,
RaycastCharacterController character) {
        if (collider.direction == RC_Direction.DOWN) {
            myCollider.enabled = false;
            myRenderer.enabled = false;
        }
    }
}
```

This platform will disappear as soon as you stand on it. Notice how the direction is checked to ensure that it is a DOWN collider (i.e. a feet collider). You can also create actions for when you hit a platform with your head or for when you run in to a platform.

Also important is ParentOnStand:

```
1.  /// <summary>
2.  /// Does this platform want to have this platform become the characters
    parent. Used for moving platforms.
3.  /// </summary>
4.  /// <returns>
5.  /// Return a transform if you want to reparent the character.
6.  /// </returns>
7.  virtual public Transform ParentOnStand(RaycastCharacterController character)
```

You can return any transform from this function and the character will be parented to the transform for as long as they are standing on the Platform. Usually you would return the transform of the parent (for example see moving platforms in the Sample scene), but in complex examples (i.e. Ropes) you may want to use a different parent.

As of version 1.2 you can do even more with your platforms. Use the **overrideX** and **overrideY** properties to completely control your character movement when they are parented to your platform. See the rope scripts for examples of this.

Check out the samples for more ideas.

Animations

Platforms also give you the ability to override animation state. Do this if you want to play a different animation on your platform (for example jump around when you are on hot coals).

If you want to create your own animation states you will also need to update the **CharacterState** enum in the **RaycastCharacterController**.

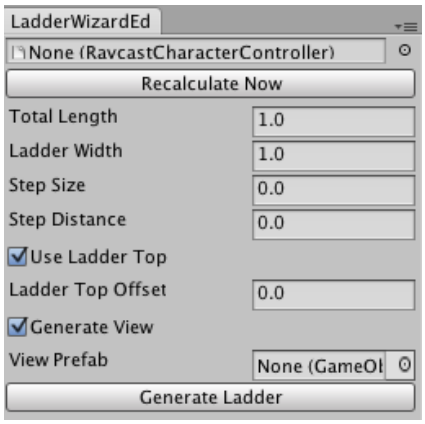
Ladders (UPDATED IN 1.7)

Users upgrading from 1.6 or below please note that the new ladders completely replace the previous ladders. If you update your project you will need to manually upgrade all your ladders.

Ladders are constructed of multiple small colliders that belong to the climbable layer. The player "stands" on each of these steps. A ladder also includes a top level construct called the **LadderControl** which controls the settings across the whole ladder.

As of version 1.7 ladders can be created with the ladder wizard which can be found in the menu system under **Assets->2D Platform Controller->Ladder Wizard.**

The first parameter is the character for which the ladder is being made. Assign a character and use the **Recalculate Now** button to automatically calculate some values.



Total Length and **Ladder Width** control the size of the ladder in world units.

Step Size and **Step Distance** control the size and spacing of the ladder colliders and can usually be calculated automatically.

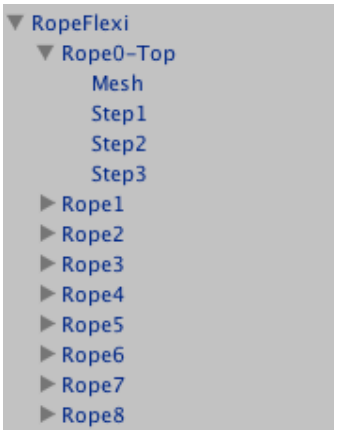
If you use a ladder top an automatic climb up (and down) state will be triggered as you get to the top of a ladder. The ladder top offset is calculated automatically from character height but may need some tweaking.

Finally **Generate View** allows you to assign a prefab that is automatically scaled to match the ladder and can be used for creating a visible component for your ladder.

There are also new ladder settings. Most are self-explanatory although be sure to turn off **Snap To Middle** and **Dismount With Arrows** if you want to enable sideways movement.

Ropes

Ropes are implemented using the platform mechanism. They have a somewhat complex structure. Parts of the rope:



RopeFlexi is the **root** of the rope. It contains the rope controller which has various rope settings. You can control the physics of the rope release here.

Rope0-Top through to **Rope8** are **rope sections**. These are standard unity physics objects. For a flexible rope they are connected by hinge joints. Theoretically you can use any configuration here but be careful Unity physics settings are very sensitive.

Step 1 through to **Step 3** are colliders on the climbable layer. They work just like rungs on a ladder. The bottom rope (8 in this case) will have an additional collider

Mesh is used to draw the rope. You can replace this with your own 2D library, or use the control points of the rope sections to draw a curved rope (for example create a mesh based line using the Vectrosity plugin).

The rope root and the rope sections **MUST HAVE UNIFORM SCALE** as the character is parented to them.

To extend a flexi rope duplicate the last section of the rope and rename it (for example to Rope9). Move to appropriate position. Set the connected body of the hinge joint to the previous rope section (e.g. Rope8). Remove the extra collider from the previous rope section.

To extend a stiff rope just increase the scale but **REMEMBER TO KEEP THE SCALE UNIFORM.**

Note that if **autostick** is false you will need to press up or down to grab on to a rope.

Triggers and Colliders

You can also add a trigger or collider to your character. In this case change the physics settings to ensure the trigger or collider doesn't interact with the platform controller layers (background, passthrough, and climbable).

If you add a kinematic rigidbody you must ensure that gravity is not applied and that the rigidbody is on a layer where it **can never** hit any other colliders.

Input

Your input class should extend the provided abstract class (**RaycastCharacterInput**).

X values control direction. A magnitude of one or greater represents a run value, a magnitude between 0 and 1 (exclusive) represents a walk value.

Y values are used for climbing and crouching

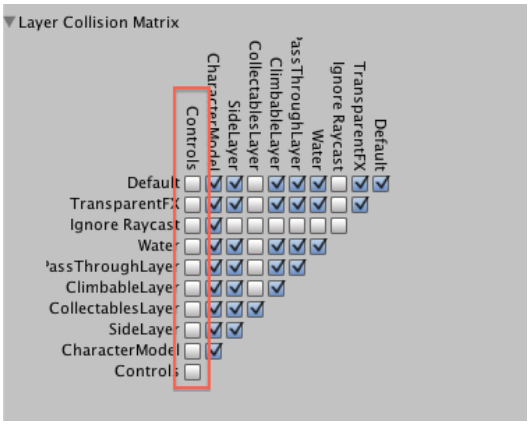
Jump and JumpHeld control the start and duration of a jump.

Touch Input

Users upgrading from 1.5 or below please note that the new touch input is generally much better than the previous sample based on Unity's touch controller. Recommendation is to upgrade.

A new touch input is available see HeroSampleWithTouch scene. Note that this control is a digital control; if you want analogue control you may still prefer to use the older controls (the StandardMobileInput class is still available).

Note that you should ensure the controls are on their own layer (for example a new layer called controls), and that the physics settings don't allow anything to collide with this control layer, for example:



Animation

Users upgrading from 1.2 or below please note that the Animation events have changed. Previously multiple event types were sent, now only one event method is sent (which contains details of the current and previous state). This change is part of a big improvement in animation but it will mean you need to change any animation controllers you have created.

Animations are sent via CharacterAnimationEvent which is a c# event delegate. By default only changes of state are sent. An animation event can also be sent every frame by selecting the appropriate checkbox in the editor settings.

See the example **ModelAnimator** for a sample of working with animations.

You can also attach the **AnimationLogger** behaviour to your character to see the animation events being sent in the console.

You can also use the **State** property of the controller to inspect animation state.

Other properties may also be useful when animating for example the Velocity property of the controller, and the input properties available from the RaycastCharacterInput.

One Frame States

Animation states like JUMPING, DOUBLE_JUMPING, and ROPE_SWING are only sent in the frame they occur. In the frame immediately after you will get a different state. For example the AIRBORNE state is sent the frame after a jump.

A normal jump looks like this:

JUMPING -> AIRBORNE -> FALLING -> IDLE

It is up to you to determine when your jumping animation stops playing. A typical scenario might be to set the end of your jump animation to look like your character is flying upwards and to attach no animation to the AIRBORNE state.

An alternative might be to have a JUMPING and AIRBORNE animation but to priorities the JUMPING animation so it plays in full before the AIRBORNE animation is played.

Working in UnityScript (Unity JavaScript)

Move the scripts to a directory called **Plugins** to ensure they are available to UnityScript (should be done by default).

Because Unity Script does not support c# events you have two options for working with Animations:

1) Write a c# class which sends the events to your Javascript code (for example using SendMessage).
2) Inspect the State and Velocity property of the controller.

Extra Features

Version 1.6 is the first release to focus on providing features that aren't part of the core platform controller. These features are provided as a learning aid to assist with understanding the controller; they may of course be used in your own projects too.
Unlike the core code is expected, even recommended, that you adjust this features to suit your own needs. That said it might be a good idea to give them a new name or place them in a new directory to avoid issues when updating.

Extra features are documented inline in the code, and demonstrated in one of the many new samples. Below is a summary of the provided features, please ensure you read the inline documentation.

Enemies

Several enemy AIs are included. It is very likely that you will need to customize these for your own games, however the code does demonstrate how you can either:

Use the full *RaycastCharacterController* for an enemy writing your AI as an input to the controller (as seen in **EnemyAIController.cs** and demonstrated in the sample **HeroSampleWithEnemies**).

Extend the *RaycastCharacterController* to create your own controller that can limit (or extend) what the core controller does (as seen in **EnemyBounceAndFall.cs** and demonstrated in the sample **HeroSampleLikeMario**).

Create simple enemies that don't use any platform mechanics (as seen in **EnemyPatrol.cs** and demonstrated in the sample **HeroSampleWithEnemies**, enable the disabled object to check it out).

Damage

Includes a basic script to track your character health (SimpleHealth.cs). This script is integrated with a **HitBox.cs** which triggers damage to your player and can be added to bones in order to move with your character. It is also integrated with **FallDamage.cs** which allows for constant or variable fall damage.

These features are demonstrated in **HeroSampleWithDamage**. Also in this sample is a very simple health bar, it is unlikely this health bar is suitable for production use, its there to make the demonstrations easier to understand.

Collectables

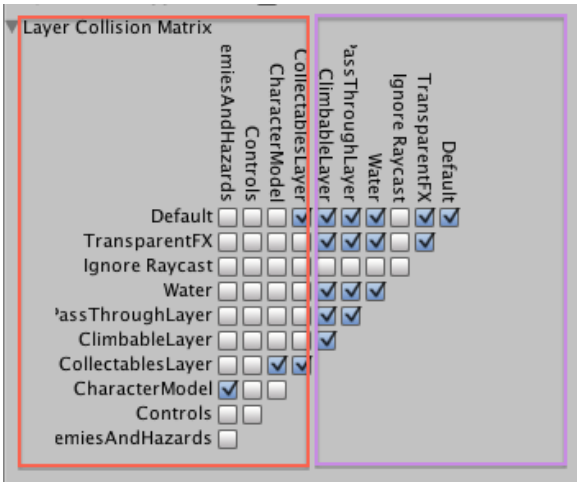
The **HeroSampleLikeMario** shows how you can create and interact with simple collectable objects like coins. It is also includes simple spawning of collectables (i.e. bricks which yield a coin when hit).

This feature relies on the **HitBox.cs** script from above and also the **Collectable.cs** script.

A note on layer setup

It is very important that the layers are set up correctly for these features to work. Specifically your character layer should only collide with the enemy and hazards layer and the collectable layer.

It's also important to ensure that objects you can't stand on or push against aren't in the background, passthrough or climbable layers (see the layers section earlier).



The purple layers are the standard layers used in previous versions.

Note how the new **red** layers collide with very little. In this case the coins are set to collide with the default layer so that coins can bounce around.

The controls layer is used for the touch controls to ensure they don't hit any in game objects.

Samples Requiring Third Party Assets

Version 1.6.2 is the first release to add samples that depend on other Asset Store packages. This version includes the AlienSample which is a platform game built for 2D ToolKit.

Like the extra features the code is included as a sample and you shouldn't consider it suitable for every game.

Versions of this sample for other popular packages are coming soon.

2D Toolkit

Steps for Use

1. Import the Platform Controller from the Asset Store.
2. Import 2D Toolkit form the Asset Store.
3. Navigate to **Samples** folder and import the package **AlienSample-2DToolKit** by double clicking it.
4. Open the sample scene **2DSample/2DToolKitSample**

Description of All Settings

This section has been removed. Description of each setting is given in the code as a <summary/> item above each property.

It is recommended you read through the inline comments.