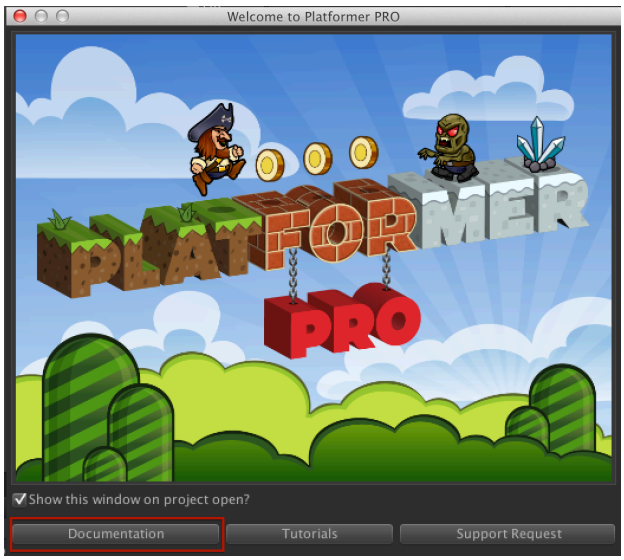
Platformer PRO

v1.1.3 - Light Edition

*This document is an excerpt from the online documentation. If you are able we recommend you use the online version of the documentation which can be found by clicking the **Documentation** button on the Welcome Pop-up screen.*



JNA Mobile

Platformer PRO - Documentation

Getting Started

How to get Help

Use the Submit a Request link on this page to create a support ticket.

or

Join our **forums** at: http://platformerpro.boards.net

View page

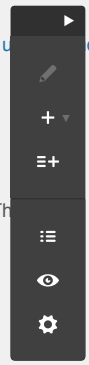
I want to get started quickly

Timing: Approximately 4 hours to complete.

For people who wish to get something up and running as quickly as possible. This is best suited to people with an **intermediate** understanding of Unity. You need to understand how to create objects, add components, duplicate objects, find and assign variables to components, etc.

If you are new to Unity you may prefer to look at the section:

I wish to u d Platformer PRO



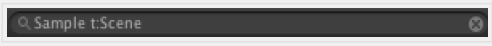
Get Started

Lets begin by creating a folder at the root of the project hierarchy. Th referred to as the *prototype folder*. You can name 'prototype', or name it for your game (or whatever you desire).

Browse the Samples

In this approach we are aiming to get something worthy of being considered a 'prototype' up and running as quickly as possible. The easiest way to do this is to find the sample thats closest to your ideas and use that character as a basis.

Search for scenes in Unity using the Project search query: "Sample t:Scene". You may want to save this search query.



Play the samples and find the one thats most similar to your game.

***Warning**: This article assumes you use a 2D sprite based character. If you select a 3D character most of the steps are the same, but setting up your animator will require you to read some of the other sections of this documentation.*

Once you have found a sample create a new prefab from the Character object in that sample (this is the *GameObject* with the *Character* script attached). Make sure this new prefab is in your prototype folder and rename it to 'PrototypeCharacter'.

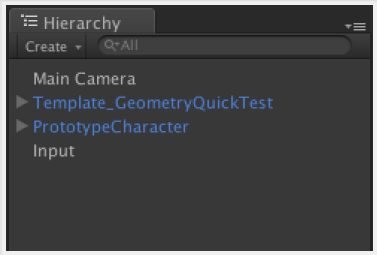
***Tip**: Its best if you break the prefab connection between the sample and your newly created prefab.*

Create Your Level

1. Create a new scene and save it to your prototype folder and rename it to 'PrototypeScene'.
2. Open this new scene.
3. You may want to find the MainCamera and change it to Orthographic projection. You may also want to set the size to a larger number like 8.

4. Drop the *PrototypeCharacter* in to the scene as a top level object and make sure its at position (0,0,0).
5. Find the prefab called '*Template_GeometryQuickTest*' and drag it in to your scene.
6. Create an Empty GameObject in your scene and rename it to *Input*.
7. Add a *StandardInput* Component to this GameObject.

Your scene hierarchy look like this:



Press **Play** and you should be able to move your character around the game world using the default keys (arrows keys, Z and X).

***Tip**: Notice how the camera and input automatically pick up the Character.*

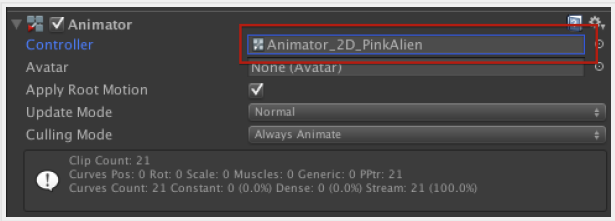
Update Your Characters Look

As a first step for 2D sprites you may want to find your characters *SpriteRenderer* and update the base *Sprite* (shown in scene view but replaced once animations start).

It's a good idea to use a neutral pose like the first frame of your IDLE animation here.

Updating Animations

Find your characters animator, it will typically be on a child of the Character GameObject called CharacterSprite or CharacterModel. With the project window visible click the **Controller** field of the Animator:



This will select the controller in the project view.

Duplicate this controller and rename it to *PrototypeController*. Move this Duplicate to your prototype folder.

Update your animator to use this new controller and **Apply** the changes to the prefab.

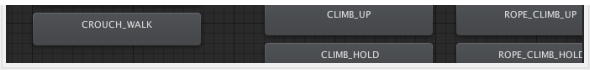
***Tip**: The steps here are done to ensure you don't alter the samples but they are not strictly necessary. You could instead just update the existing animation controller.*

For Simple Animation Bridges

With your character sprite/model selected open the Animator window. You should see the animation state machine for your character.

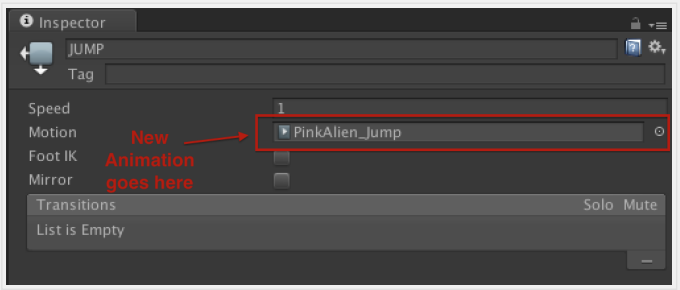
You may want to read about Animation Bridges before proceeding.

If your animator looks something like this (with no transitions between states):



your animator is of the simple type.

To update an animation simple select the Animation State in the Animation state machine and assign a new animation:



To add new animations (i.e. for different states) create a new state and make sure the name is an exact uppercase match for the state name found in the script file *AnimationState*.

***Tip**: If your new movements (see below) send extra states you will get a warning that they are missing in the editor window.*

For Complex Animation Bridges

If your animation bridge is more complex type then it is not covered in this section. Find the matching article in the Animation category of this documentation.

Colliders

If you press Play you will probably find that your characters colliders no longer match the characters shape. It might be standing well above the ground, well below the ground, or maybe it can walk inside a wall.

To adjust your colliders click your character object and press the large **Reset Colliders** button. Click the **Reset** button next to the option '*Use basic sprite detection'*.

***Tip**: If your sprite is readable you will also have the option to 'Use smart sprite detection' you are welcome to give this option a go. It tries to use transparency to better match your characters shape but wont always be accurate.*

This should give you colliders that basically match your sprite. You may want to make the following adjustments:

1. Click **Edit Feet** and then add 1 or 2 more feet colliders using the **Add Collider** button. This will help with slopes and also stop your character from falling through very thin pieces of ground.
2. Click **Edit Sides** and use the **(+) icon** in the scene view to add more side colliders. Remember if the character runs in to a wall but it doesn't hit a side collider they will walk straight through. So you need enough side colliders to ensure the gap between them is smaller than the smallest platform height. For most geometry 1 or 2 is enough.
3. Click **Edit Head** and then add 1 or 2 more head colliders using the **Add Collider** button. This will stop your character from jumping through very thin pieces of ground.

Fine tune the Colliders

Press **Play** and see where your character is standing. If he is standing too high, go back to the scene view click **Edit Feet** and use the handles in the side view to raise the Feet Colliders. If it is standing too low, do the opposite.

Repeat until it looks right.

Use the same process for the side colliders and head colliders. Pressing Play and adjusting the colliders to ensure the character meets the wall or roof in the right place.

Once done save your scene and **Apply** the changes to your character prefab.

***Note**: This process can be a little tedious but only needs to be done once for each character.*

Hit Boxes and Hurt Boxes

The *HurtBox* is the part of the character that registers damage. Like the colliders, after you update the sprite the hurt box may no longer be aligned with the character.

The HurtBox is a standard Unity Collider2D. Drag the HurtBox GameObject until the collider is in the right place. Then use the collider settings to adjust the size of the collider.

The HurtBox is also generally used for collecting items although you can use a different collider with a *CharacterReference* attached if you prefer (for example if your character can only take damage when hit in the head, but can collect items using any part of their body).

The *HitBox* component causes damage to others (enemies) and is typically associated with melee attacks. Like the HurtBoxes these are standard colliders that can be moved in to position to match your character sprite.

Customise Movement

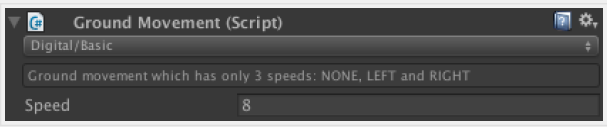
There are many, many options for customising movements. Here we discuss this using a few examples, but its recommended that you read the article:

How Movement Works

Example: Increasing the speed of the Alien Character

A simple example, we want our alien character to move faster.

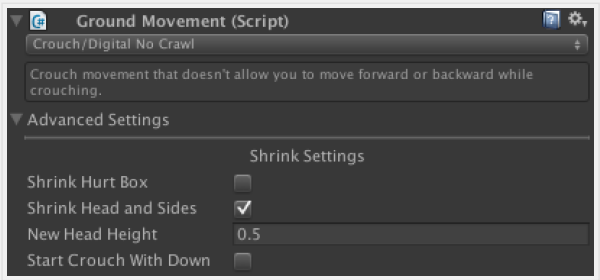
1. Select the GameObject that contains the movements for the character. For the Alien its the child object called *Movements* (for some other characters it may be the Character GameObject itself.
2. Find the Ground Movement.
3. Increase the value of the speed variable:



Example: Adding crouch to a Character

Crouch is a movement that overrides the default ground movement when the user presses down. To override a movement the movement must be placed **higher** in the component list.

1. Add a new *GroundMovement* to the Movements object.
2. Select **Crouch Movement/Digital no Crawl**.
3. Open the advanced settings and click the **Shrink Head and Sides** box.
4. Enter a **New Head Height** of 0.5.
5. Move the Crouch ground movement to be above the existing GroundMovement in the list (to do this click the gear icon in the top right of the component inspector and select the **Move** Up option).
6. Apply changes to the prefab.



***Important**: If you are using a prefab you **MUST** apply the changes to the prefab for this to work. Unity has a bug where the prefabs component order overrides the component order of the GameObject (unlike any other time when the GameObjects settings override the Prefabs).*

Add Missing Components

The prefab you based your character on may not have all of the behaviours you want. This section covers a few components you may want to add.

Health

If you want your character to take damage or just to be able to die, respawn, etc, then it needs a *CharacterHealth*.

To set it up, add the *CharacterHealth* component to the same gameObject as your Character.

There are a lot of settings here so its best to read about them in the article:

Character Health

Items

In order to collect items a Character needs an ItemManager.

To set it up, add this *ItemManager* component to your Character GameObject.

No additional setup is required for basic behaviour but you may want to read the article:

Items

Power-ups

Power-ups a re special kinds of items which give your character additions abilities. PowerUps require a PowerUp Responder to work.

To set it up, add the *PowerUpResponder* component to the same gameObject as your Character or to a child GameObject of your character

There are a lot of settings here so its best to read about them in the article:

Power Ups

View page

– I wish to understand Platformer PRO

Timing: 8+ hours to complete.

For those who want to get a better grasp of Platformer PRO before starting their own project this article suggests a path of study which takes a bit longer than the quick start.

Don't worry its called study but it should be mostly fun!

Initial Reading

A lot of the articles in this documentation are marked with an (I). Its a good idea to read through all of these as a starting point. Even if some things don't make sense right now, as you move through the tutorials things will become clearer.

Characters (I)

Colliders (I)

Input Basics (I)

How Movement Works (I)

Animation Bridges (I)

Platforms (I)

Intro Tutorial

There are a set of video tutorials available on Youtube which cover building a game from the ground up. The list is growing over time but currently includes:

Intro Tutorial - Part 1

Intro Tutorial - Part 2

Join the Forum

http://platformerpro.boards.net

Quick Start

Armed with the knowledge you have gained you may want to go through the quick start:

Quick Start

Additional Tutorials and Walkthroughs

Check the Youtube playlist here for more tutorials and walkthroughs of specific features:

Tutorial Playlist

Read Everything

you should be well on your way to understanding Platformer PRO, the next step is to read everything ... including the Full API Documentation if you are feeling brave!

Build Stuff

The best way to learn is to do, so get building!

View page

+ Full API Documentation

Character Components

− Character (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO.

The *Character* components is the base component which orchestrates all of the other components: it makes them work together to realise your character.

The *Character* component contains configuration which affects key elements of the behaviour such as what slopes and what layers the character will interact with.

To create a new character add the *Character* component to an empty GameObject.

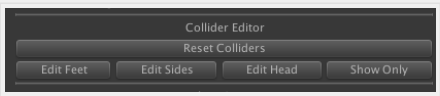
View page

− Colliders (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO.

Character collisions in Platformer PRO are controlled by the RaycastColliders you define on your character. There are a number of colliders for each of the head, sides and feet of the character.

After you create a character by adding the Character component you can define the colliders for your character using the Collider Editor:



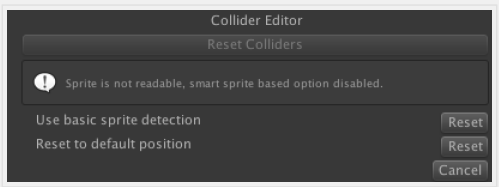
Resetting Colliders

Press the **Reset Colliders** button in order to reset your colliders back to different defaults. When you press this button you will be shown a list of options based on the configuration of your character.

Option	Details	Requirements
Reset to default position	Resets colliders to a box shape.	None.
Use basic sprite detection	Resets colliders to the sprite bounds.	Child with a SpriteRenderer
Use smart sprite detection	Reads the alpha channel of the sprite to guess the best places for the colliders.	Child with a SpriteRenderer that points to a readable Sprite.
Use mesh based detection	Resets colliders to the bounds of the mesh.	Child with a MeshFilter.

Reset collider options with a readable sprite as a child of the Character GameObject:

Reset collider options with a non-readable sprite as a child of the Character GameObject (notice the info message):



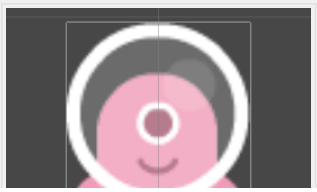
After your basic colliders are set you can use the Edit Feet, Edit Sides, and Edit head button to refine the position of your colliders.

Editing Colliders - Feet

The feet colliders push the character upwards, they are what the character ‘stands’ on.

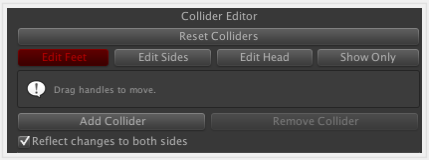
***Remember**: the colliders define raycasts, not a box collider or line. If the feet raycasts are not colliding with anything the character will fall!*

Clicking **Edit Feet** shows edit feet scene view:



Use the handles to drag the boundaries of the characters feet.

In the inspector you will see the edit feet options:



Use the **Add Collider** button to add more colliders to the character. Each collider will be indicated with a small circle drawn along the line. Use the **Remove Collider** button to remove additional colliders. Because feet colliders must be evenly spaced you can only control the boundaries of the feet colliders.

More colliders will allow your character to stand on very thin platforms. You should also use more colliders (4 or 5) if you are dealing with steep slopes. You should rarely need more than 5 feet colliders.

The **Reflect changes on both sides** checkbox keeps your character symmetrical. In most cases you want your character to be symmetrical but if you do have an asymmetrical character see Working with Asymmetrical Characters.

Editing Colliders – Sides

The side colliders push the character to the side: they stop it walking through walls.

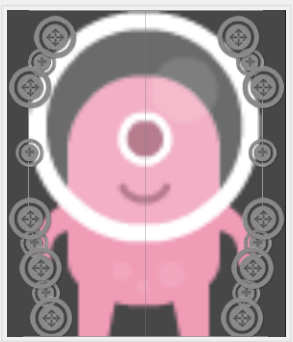
Working with side colliders is similar to working with feet colliders, however you have more control over the position of the side colliders:

Use the handle to move the colliders and the button to add a new collider. Hold SHIFT and click the button to remove a collider.

***Remember**: you need to have enough side colliders such that the largest distance between any colliders on the same side is smaller than your thinnest wall or platform. This ensures the character can't 'pass-through' the wall. Usually 3 or 4 is adequate.*

Remember how the colliders work, they push the character to the side, if your colliders are not aligned in x your character may jerk oddly as the player pushes towards walls when falling.Press **Align in X** to align all of the colliders along the x-axis. Most characters should have their colliders aligned along the x-axis.

However non-aligned side-colliders can work for some characters, particularly oddly shaped or 2.5D characters. If you want to use non-aligned side colliders aim to use many side colliders to provide a smooth shape:



The Space Evenly button will space colliders evenly along the y-axis.

The **Reflect changes on both sides** checkbox keeps your character symmetrical. In most cases you want your character to be symmetrical but if you do have an asymmetrical character see Working with Asymmetrical Characters.

Editing Colliders – Head

Head colliders stop your character from jumping through the roof.

Working with head colliders is similar to working with side colliders. Instead of aligning in x and spacing in y you have options to align in y (**Align Top**) and space in X (**Space Evenly**).

Like the other colliders you must ensure the head colliders are close enough together that the character cannot passthrough a platform.

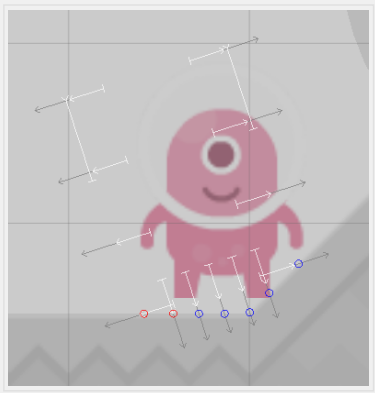
General Collider Configuration

The following guidelines will help ensure your character behaves as expected without odd pops or jerks: only break them if you are sure you know what you are doing!

- Side colliders must be inside the bounds in y defined by the head and feet colliders (this is enforced by the editor).
- Generally side colliders should 'stick out' at least a little further than the head and feet colliders.
- Generally head and feet collider boundaries should be roughly the same: if a character falls directly down from a platform they should generally be able to jump back up without hitting their head.

Debugging Colliders

You can see exactly what is going on with your colliders by pressing the Debug Collider button found on the Character inspector. In Edit Mode this shows the colliders boundaries and extents but in Play mode it shows much more:



White lines show the colliders *extent* with the arrows indicating direction.

Grey lines show the look ahead: this is the total distance the ray is cast. Look ahead is used for things like detecting walls or detecting the ground *before* pushing off of it.

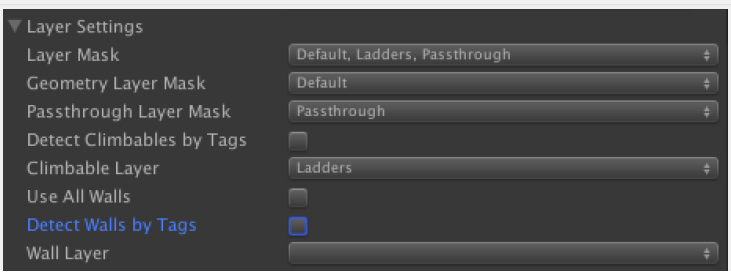
Blue circles show collisions in the look ahead zone. They may be used to decide on actions but they will not affect character position.

Red circles show collisions in the colliders *extent*. They will affect the characters position.

View page

– Character - Layer Settings

Use the Layer Settings foldout to define which layers your character will interact with:



Each field is a LayerMask which defines a set of layers that are used by the controller. The following layer masks can be set:

Setting	Description
Layer Mask	All layers that the collider can interact with must be included here. The Geometry and Passthrough LayerMasks are always present here. You may want to add ladders or other special layers.
Geometry Layer Mask	Layers of any of your games solid geometry. That is to say the things your character can stand on and run in to.
Passthrough Layer Mask	Layers of any platforms that should be treated as passthrough platforms. Passthrough platforms can be stood on but not collided with from the side or bottom.

The following additional settings are only shown if your character has a Climb movement:

Setting	Description
Detect Climbables by Tags	Should we use a tag or a layer to detect if something is climbable. Layer based detection is slightly faster and generally recommended.
Climbable Layer/Tag	If we are using layer based climbable detection this is the layer that the climbable belongs to. If we are using tag based climbable detection this is the tag to check for.

The following additional settings are only shown if your character has a Wall movement:

Setting	Description
Use All Walls	If true then all walls will trigger the wall movement(s).
Detect Climbables by Tags	Should we use a tag or a layer to detect if something is a wall. Layer based detection is slightly faster but tag based detection can be easier to manage.
Climbable Layer/Tag	If we are using layer based wall detection this is the layer that walls belong to. If we are using tag based wall detection this is the tag to check for.

View page

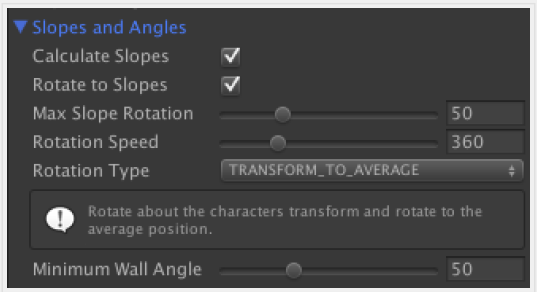
– Character - Slope Settings

Slope settings allow your character to walk on sloped surfaces by rotating the colliders to match the surface.

You can also support slopes by simply snapping to the ground and not rotating. This is more typical in old school pixel based games. See Character - Supporting Slopes without Rotating.

***Remember**: Even if you do rotate your controller you may still keep your character sprite or model rotation free by using a DontRotate script.*

The following slope controls are available:



Setting	Description
Calculate Slopes	If this is on the character will calculate the angle of slope it is standing on. This will be available via the property *SlopeTargetRotation*(). You may way to calculate slopes without rotating to, for example, drive sliding behaviour.
Rotate to Slopes	Should the character transform rotate to the angle of the slope. Note that you do not have to enable this to enable slopes see Supporting Slopes without Rotating. This setting is generally used for 2.5D characters or characters where the alternate method is not adequate. If your character rotates to slopes it does not mean your sprite needs to rotate. You can add a *DontRotateSpriteScript* to your sprite to support this.
Max Slope Rotation	How far is your character allowed to rotate from identity (0). Note that although large values will allow you to run up walls or even upside down not all movements support this. See the section on Loops for more details.
Rotation Speed	How fast will the character move to the target rotation. If your sprite does not rotate you should use a very large value (e.g. 720). Smaller values will mean your character slowly rights themselves when jumping and can cause issues on steep curves.
Minimum Wall Angle	The minimum angle at which a piece of geometry can be considered a wall. This is a relative measure and is calculated based on the normal of the collision between a side collider and the geometry.

View page

Character - Supporting Slopes without Rotating

If you are creating a game in a retro-style you may want to support slopes without rotation. This approach uses the colliders to push the character upwards when they intersect with a slope, so the character will always stand on top of the slope.

When moving down a slope this approach requires a movement which supports **Snap to Ground**. Snap to ground support means the movement will automatically move the character down to ensure its feet are aligned with the ground.

At the time of writing the following movements support Snap to Ground:

- Digital
- Digital with Run
- Physics (optional setting)

In order for this to work on steep angles you need to ensure your Characters Grounded Look Ahead (set in Character - Advanced Settings) is large enough to always detect the ground. You can calculate this using the equation:

*sin(maxGroundAngle) * maxSpeed * maxFrameTime = grounded look ahead*

For example if your character moves at a max speed of 8 units per second and you want to support 30 degree angles you need a lookahead of (assuming the default max frame time of 0.033):

0.5 * 8 * 0.033 = 0.132

You can see this setup in action in the scene: **SampleScene-Alien-Level1.1**

View page

Character - CharacterReference

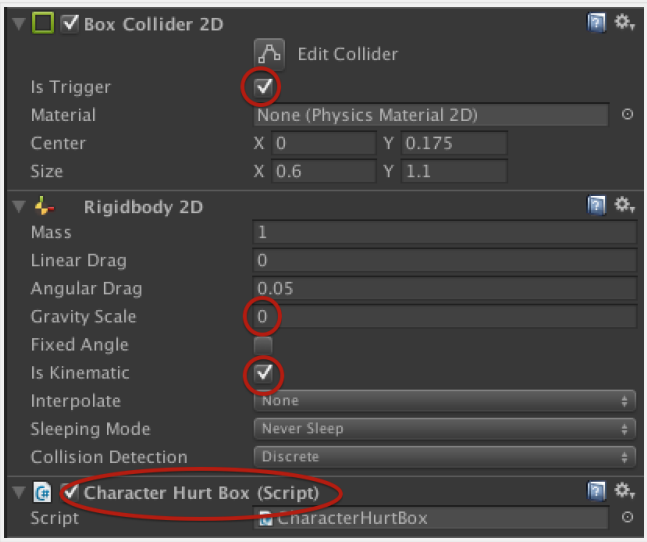
In order to interact with objects in the scene (e.g. Some type of Triggers, Items, Enemies) your character needs a CharacterReference.

A HurtBox serves as the CharacterReference for most Characters,

TheCharacterReference component is typically added to a child GameObject of the characters model or sprite. This way it will flip when the model or sprite is flipped.

This component also requires some kind of *Collider2D* (for example a *BoxCollider2D*) and because it moves should also include a *Rigidbody2D*.

The Collider2D **must** be a trigger, the Rigidbody2D **must** be kinematic with a gravity scale of 0:



Layers

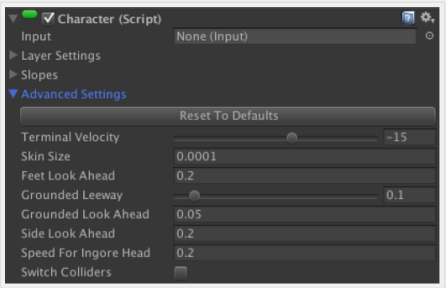
The default layer for a CharacterReference is the *Character* layer.

View page

– Character - Advanced Settings (A)

(A) ADVANCED: This section is aimed at advanced users.

Advanced settings control the fine details of the characters movement, adjust with care. The following options are available:



***Remember**: If you get stuck press the Reset To Defaults button to bring back the original values.*

Setting	Description
Terminal Velocity	Hard limit on how fast your character can fall. Must be smaller than 0. For a 1 to 2 unit high character a value of -15 to -30 is suggested.
Feet Look Ahead	How much further, beyond the bottom of the feet, to cast the ground colliders. Effects ladder detection, landing detection, etc.
Grounded Leeway	The time after leaving the ground for which the character is still considered to be grounded. Primarily used to allow the character to jump just after it has ran off the edge of a platform.
Grounded Look Ahead	How much further, beyond the bottom of the feet, to check when considering the character to be grounded. Generally leave the default unless your character is reporting as not grounded when they should be.
Side Look Ahead	How much further, beyond the sides of the character, to cast the side colliders. Primarily affects wall and ledge detection.

Speed for Ignore Head	The downward speed at which head collisions are completely ignored. Lower values give smoother movement, but wont work with fast moving vertical platforms.
Switch Colliders	Should we switch colliders when the character changes direction (used for asymmetric characters). It is best to fix symmetry issues in the model or sprite however sometimes this is either not possible or not practical. See Asymmetric Characters.

View page

Input

– Input Basics (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO.

This section covers the basics of controlling a Character via an Input. An Input is any concrete implementation of the abstract *Input* class.

Quick Start

To quickly add an input to your game:

1. Create an empty GameObject in your Scene.
2. Select your newly created GameObject
3. Press the **AddComponent** button.
4. Add the StandardInput component to your GameObject (you can quickly find the component by typing its name).
5. (Optional) You may want to press the **Clear Preferences** button on the Standard Input to ensure there are no Player Preferences set.

Overview of Input Classes

The following Input classes come with Platformer PRO:

Input Class	Description	Link
Standard Input	The standard input is a flexible and powerful input class for desktop and consoles. Most games that aren't for mobile will use the StandardInput.	Standard Input
Basic Touch Input	A simple input for touch devices.	
Unity Input	An Input class that allows you to use standard Unity input definitions.	
Buffered Input	A special input class, which buffers the input from another input to provide more sensitive controls. You cannot use a buffered input by itself.	Buffered Input
	An input class which provides a single	

Multi Input	interface in front of multiple other inputs. For example it allows you to have both keyboard and joystick configured simultaneously.	Multi Input

How a Character find its Input

Although you can directly assign an Input to a character using the character s *Input* property if this is not assigned the character uses the following process to find an Input:

1. Find all active Inputs in the scene.
2. If none are found log an error.
3. If one is found assign it.
4. If two or more are found:
    1. Try to find a Buffered Input and assign it.
    2. If no Buffered Input is found, log a warning and assign the first Input found.

Action Buttons

Buttons like Jump and Run, Up and Down, are self explanatory but you may also need extra buttons in your game for things like using items or attacking. Action buttons allow you to define any number of arbitrary buttons.

When used in other components action buttons are identified by their array index. So for example if you define two buttons as follows:

And you want to use the second button (V) to attack you would set the action button index of the Attack to 1.

***Tip**: As with C# arrays are zero-indexed. In other words the first action button has the array index of 0.*

Using Third-Party Input Systems

You can use other Input systems with Platformer PRO by wrapping them in your own implementation of the abstract class. For example if you were using cInput 2 (a popular third party Input asset) you might write a class as follows:

```
public class cInputWrapper : PlatformerPro.Input
{
  public string jumpButtonCInputName = "JUMP";
  override public ButtonState JumpButton
  {
    get
    {
      if (cInput.GetKeyDown(jumpButtonCInputName)) return ButtonState.DOWN;
      if (cInput.GetKeyUp(jumpButtonCInputName)) return ButtonState.UP;
      if (cInput.GetKey(jumpButtonCInputName)) return ButtonState.HELD;
      return ButtonState.NONE
    }

    // And so on for the rest of the methods ...
  }

}
```

Unused Methods

Note that the Input class defines a number of abstract methods which allow in-game customisation of input. In many cases if you are using a third-party system these may not be applicable. Suggested implementation in these cases is something like:

```
override public KeyCode GetKeyForType(KeyType type, int keyNumber)
```

```
{
  Debug.LogError ("This Input does not allow for in game control configuration.");
  return KeyCode.None;
}
```

View page

– Standard Input

The Standard Input provides a full featured Input for Keyboard and Controller. It includes the ability to configure the input in-game and to load preferences from presets defined in a file.

How It Works

The Standard Input loads data from Player Prefs when it is initialised. If there is no data in the Player Prefs then the input will use the **Default Controls** defined for the GameObject.

Data To Load

The Data to Load configuration allows you to define a unique ID for the input. This ID will be used to load the data from Player Preferences when the input is initialised.

You can generally leave the default value unless you are creating a multi-player game in which case you should set a different ID for each player.

Default Controls

> ***Remember**: The default controls are only used if there is no data defined in the PlayerPrefs (i.e. when the player first loads up your game).*

By expanding the **Default Controls** collapsible you can configure the default inputs which will be used if the player has not set any preferences

Axis Configuration

If Controller is enabled then the following axis will be used:

Setting	Description
Horizontal Axis	Left/Right axis used for moving back and forth.
Vertical Aix	Up/Down axis used for climbing ladders, crouching, etc.
Alt Horizontal Axis	An alternate horizontal axis which controls something other than movement (for example you could use it for aiming like in the Pixtroid Sample variation).
Alt Vertical Axis	An alternate vertical axis which controls something other than movement (for example you could use it for aiming like in the Pixtroid Sample variation).

Other Axis Settings

Each axis has a number of other settings:

Setting	Description
Reverse Axis	Should the axis be reversed (i.e. does -1 equal right instead of left). Often set as part of in-game controller configuration.
Digital Threshold	At what point does the analog axis value set the digital value for the axis.

Axis Naming

All axis names follow the pattern:

Joystick<Number>Axis<Number>

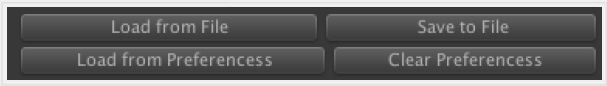
So for example the first joystick left stick horizontal axis might be called:

Joystick1Axis1

***Warning**: These axis names **must** be configured in the Unity Input settings. If you correctly import Platformer PRO these axis should be automatically created.*

Working with Input Data

There are several buttons which allow you to work with the Input Data:



Button	Usage
Load from File	Load input data from an input XML file and set them as defaults.. Several XML files are already defined with various input configurations.
Save to File	Save the currently configured defaults to an XML file.
Load from Preferences	Load the values currently set in PlayerPrefs and set them as the defaults.
Clear Preferences	Clear the Input Player Prefs.

***Note**: The saved input data files can be used as presets that a user can load in-game.*

See also: In-game Configuration of Input

View page

− In-game Configuration of Input

Platformer PRO supports the in-game configuration of Input. The Standard Input includes a full implementation of this behaviour and many of the samples include this capability.

How it Works

The Input class defines a number of methods for changing the Input Data which binds keys to actions. The implementation of these methods is up to the individual implementing class.

To perform in-game configuration you simply call the relevant Input method, either form your own code or by using the corresponding UIMenuItems from the Menu System.

Because implementations can be quite different in this section we primarily discuss how the Standard Input supports in-game configuration.

***Warning**: not all Inputs support in-game configuration.*

Setting a Key

Keys are set by calling the method:

```
override public bool SetKey(KeyType type, KeyCode keyCode)
```

This method maps the KeyType to the Unity KeyCode. KeyType must not be an axis.

Setting an Axis

Keys are set by calling the method:

```
override public bool SetAxis(KeyType type, string axis, bool reverseAxis)
```

```
override public bool SetAxis(KeyType type, string axis, bool reverseAxis)
```

This method maps the KeyType to the given axis. KeyType must be an axis.

Saving the Current Settings

To save the input data call the method:

```
override public bool SaveInputData()
```

The Input abstract class does not enforce any definition for "saving". In the StandardInput "saving" means writing the data to PlayerPrefs.

Using the in-built Menu System

In order to allow the easy creation of a menu for in-game configuration of input you can use the Menu System. The following UIMenuItem types are available:

Item	Usage
UIMenuItem_KeyConfig	Use this to set the value for a given key. Simply set the KeyType (and optional the action button index if this is for an Action Button).
UIMenuItem_SwitchInputType	Load input from the Input Data file. Contains a list of human readable names mapped to file names. Note that data files are expected to be defined in a Resource folder

Configuring the Menu

In order to use the in-game configuration Menu Items you must:

1. Configure a Menu in your scene (see:).
2. Create an Input in your scene (for example add a StandardInput component to a GameObject)
3. Create a sub-menu for key configuration.
4. Create KeyConfig menu items for each key that needs to be configured.
5. Create a SwitchInputType if you want the user to be able to cycle through presets.

See the **CommandBro** and **Pixtroid** sample scenes for an example of in-game configuration of input.

View page

– Buffered Input

The BufferedInput class allows you to buffer a key press (e.g. the jump button) such that the button down state (ButtonState.DOWN) is recorded not just for one frame but for multiple frames.

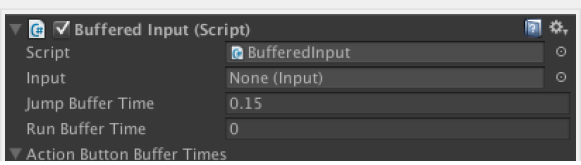
This is typically used to allow the player to jump even if they press the jump button a little before they land on a platform.

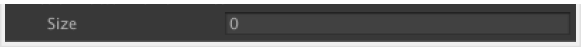
***Note**: If you want to allow the player to jump a little after they fall off a platform see the jumpButtonLeeway property found in most AirMovements.*

How To Set Up

The easiest way to set up the buffered input is to add the BufferedInput component to the same GameObject or a parent GameObject of your standard Input component:

The BufferedInput has the following settings:





Setting	Description
Input	Input that these buffer settings apply to. This is optional, if it is not present the buffered input will try to find an input on the current or a child GameObject.
Jump Buffer Time	How long to buffer the jump button for.
Run Buffer Time	How long to buffer the run button for.
Action Button Buffer Times	List of buffer times for each action button. The array index **must** corresponds to the array index of the action button it is buffering.

View page

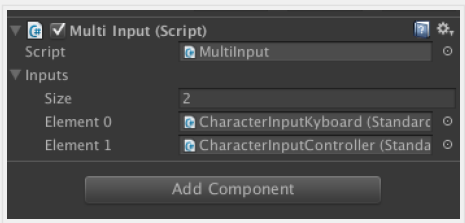
− Multi Input

The MultiInput is an Input which accepts input form multiple sources and provides an interface that makes them look like one source. You can use it configure multiple inputs for one character simultaneously, for example having both a keyboard and joystick input active at the same time.

***Note**: You can see the MultiInput in the CommandBro Sample.*

How To Set Up

1. Add the MultiInput component to a GameObject in your scene and then add references to your other inputs in the *Inputs* list:



***Note**: The order of inputs in the list controls their priority. A higher input will take priority over a lower input.*

2. Once the Multi Input is setup assign it to your character.

View page

Movement

− How Movement Works (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO.

Platformer PRO includes a flexible movement system which allows for new movements to be easily added to an existing character.

Each movement added to the Character *GameObject* OR any child *GameObject* of the Character *GameObject* is added to an ordered list of movements called the **movements list**. The order that Movements appear in the *GameObjects* is the order in which they are added to the list. Those highest in the list of *Components* in the *GameObjects* are added to the list first and have a higher priority than Movements added later.

Default Movements

The character requires at the very least two default movements. One for when in the air and one for when on the ground.

The last *GroundMovement* in the movement list is considered to be the *DefaultGroundMovement*. The last *AirMovement* in the movement list is considered to be the *DefaultAirMovement*.

Determining the Current Movement

Each frame the character works out which movement should be controlling the character at this point in time. If the character is on the ground then they will probably be controlled by a GroundMovement, if they are climbing a ladder then they will probably be controlled by a LadderMovement, and so on.

It is important to understand how this decision is made:

Step 1. At the start of each frame the **activeMovement**, that is the movement that was in control last frame is asked if it *WantsControl()*. The activeMovement is always given this option to maintain control (see Defining Custom Movements for more information).

Step 2. If the activeMovement does not want control then each Movement in the movement list is then queried to see if it wants control. The nature of the method depends on the movement type (for example an *AirMovement* will be asked if it *WantsJump*() or *WantsAirControl*()).

Step3. If none of the movements wants control then either the *DefaultAirMovement* is given control if the character is in the air or the *DefaultGroundMovement* is given control if the character is not in the air.

Transitioning Movement Control

Once the movement for the current frame is determined the character will check to see if the new movement is different to the active movement. If they are not different nothing special happens and the activeMovement continues to control the character. If they are different the character transitions the movement control over to the new movement:

Step 1. The activeMovement is informed that it is *LosingControl*(). This enables the movement to clear any data and reset the state back ready to gain control at a later time.

Step 2. The new movement is told that it is *GainingControl*(). This enables the movement to do any initialisation required such as transforming the current velocity in to a measure that is understood by this controller (see Understanding VelocityType).

Step 3. The new movement becomes the active movement and takes over control of the character.

An Example

Imagine a character with the following movements in this order:

1. GroundMovement - Crouch
2. AirMovement - Digital (DefaultAirMovement)
3. GroundMovement - Digital (DefaultGroundMovement)

If the player is standing on the ground no movement wants movement control and thus control falls through to 3 because it is the default.

If the player then presses the jump button the AirMovement (2) will return true to *WantsJump*() and thus control will be passed to 2.

If the player presses down while in the air nothing happens the GourndMovement (1) will not want control because the character isn't grounded. The control will default back to the DefaultAirMovement (2).

Once the player continues to hold down when the character lands on the ground control will now pass to the Crouch movement (3) because it *WantsGroundControl*() if the character is grounded and the player is pressing down. The character will crouch.

Now consider what happens if we switch the order of 1 and 3.

1. GroundMovement - Digital
2. AirMovement - Digital (DefaultAirMovement)
3. GroundMovement - Crouch (DefaultGroundMovement)

If the player is standing on the ground no movement wants movement control and thus control falls through to 3 because it is the default. The character will crouch instead of idling!

Jumping will still work as normal but the character will never be able to stand.

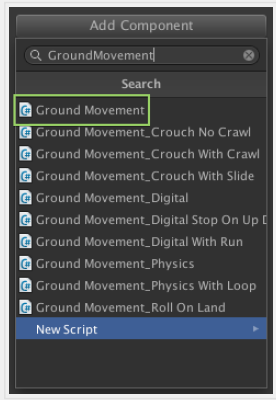
***Important**: If you are using a prefab for your Character you **MUST** apply the changes to the prefab for the component order to be applied. Unity has a bug where the prefabs component order overrides the component order of the GameObject (unlike any other time when the GameObjects settings override the Prefabs).*

View page

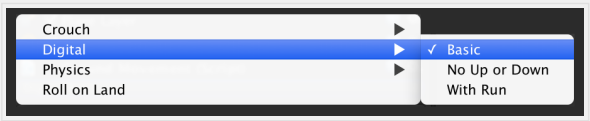
– Ground Movement

In order to move on the ground the character needs a *GroundMovement* component. A character MUST have at least one *GroundMovement*.

There are many GroundMovements supplied with Platformer PRO but you should always add the base *GroundMovement* component called *GroundMovement*. This provides a custom Inspector which you can use to customise your movement:

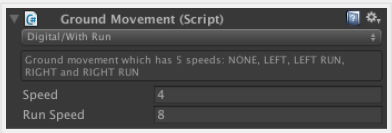


Once this is added you can use the custom inspector to browse through available ground movements:



***Remember**: If you follow the coding conventions then your own movements will appear in this menu too.*

Selecting a movement provides details of the movement and gives you an inspector which you can use to customise the movement:



Ground Movement Types

The list of ground movements is not fixed, here we detail two of the basic ground movements. See information, tooltips, and code documentation for more details on individual Ground Movements.

Ground Movement - Digital - Basic

`Ground movement which has only 3 speeds: NONE, LEFT and RIGHT`

The simplest of ground movements, this movement has only one variable which is the characters ground movement speed.

Ground Movement - Physics - Standard

`Ground movement which uses drag and acceleration to give physics like movement.`

The standard physics-like implementation from which most others extend. It has a number of settings for controlling how the physics works:

View page

– BaseCollisions (A)

(A) ADVANCED: This section is aimed at advanced users.

The BaseCollisions component controls the basic interactions between a character and the level geometry. It stops the character falling through floors, running through walls and jumping through platforms. It uses the *RaycastColliders* and layer settings to calculate this behaviour.

Some movements do not apply base collisions instead applying their own rules. See Writing Custom Movements.

The BaseCollisions component has no settings and for most users will never need to be considered. However you can override the BaseCollisions class and provide your own behaviour if you have very unique requirements.

Warning**: Customising the BaseCollisions can have wide reaching and hard to determine ramifications on the system. **Handle with care!

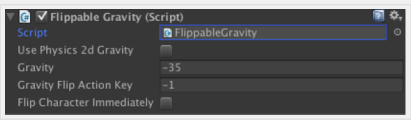
View page

– Gravity

The Gravity component provides settings for controlling gravity. The standard gravity component applies gravity in the world y-axis and provides the following settings:

Option	Details
Use Physics 2d Gravity	If checked the *Phsyics2D.gravity.y* value will be used for gravity.
Gravity	Value to use for gravity (ignored if Use Physics 2d Gravity is checked).

There is also an alternate Gravity component provided called *FlippableGravity*. This component allows gravity to be flipped from a negative y value to a positive y value. It provides the following settings:



Option	Details
Use Physics 2d Gravity	If checked the *Phsycsi2D.gravity.y* value will be used for gravity.
Gravity	Value to use for gravity (ignored if Use Physics 2d Gravity is checked).
Gravity Flip Action Key	Index of an action Key in the Input which triggers the gravity flip. Use -1 if you want to control the gravity flip from triggers, code or another means via the public method *FlipGravity()*.
Flip Character Immediately	If set the character will be immediately rotated by 180 degrees when the gravity is flipped. If unset you instead rely on the movements to set the orientation for the character.

***Warning**: Not all movements support flippable gravity. The movement inspector will* warn you if a movement has not been written to specifically support flippable gravity.

View page

– Ledge Climbing

LEDGE CLIMBING IS NOT AVAILABLE IN THE LIGHT EDITION

Ledge climbing allows characters to grasp and then climb up on to ledges.

This article refers to the ledge climbing capabilities of the movement **Wall Movement->Ledge Climbing/Standard**. Other types of ledge climbing are available via (for example) **Special Movement->Mecanim** but they are not covered in this article.

An older video (pre-release but still mostly accurate) walking through 2D ledge climbing is available here (Youtube).

Controls

Ledge climb has the following controls:

Action	Control
Grab Ledge	Face towards ledge (not a control, but you wont grab a ledge you aren't facing so you may need to press towards it).
Climb ledge	Press up while grasping ledge.
Dismount Ledge	Press down, away or jump while grasping ledge.

You could extend the class and override the following methods to change the controls:

WantsCling()

UserPressingClimbKey()

UserPressingFallKey ()

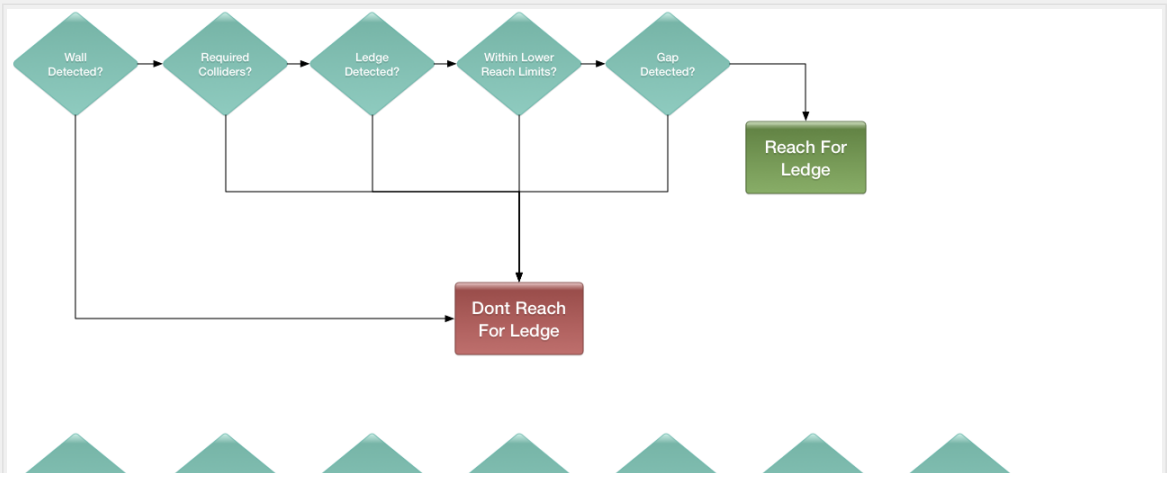
Ledge Detection Overview

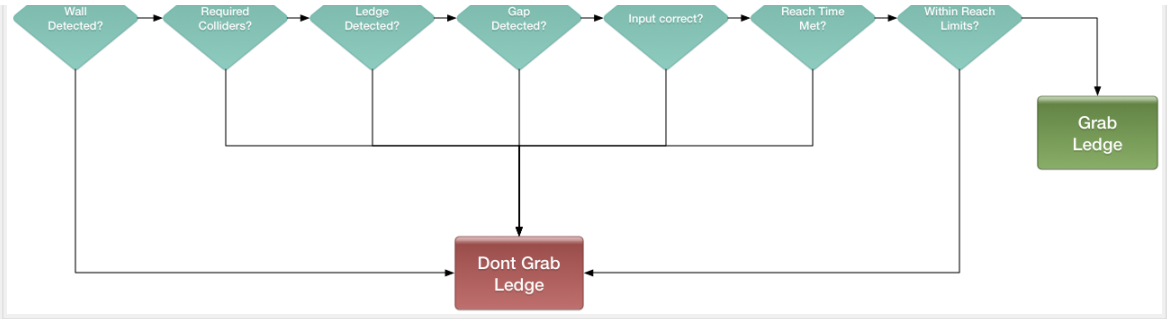
In order to determine if the character should reach out for or grab on to a ledge a number of conditions need to be met. When setting up your ledge climb its important to understand these as a failure at one point will cause the ledge climb to fail.

***Tip:** Coming soon (target v1.1.0) is a debug window which shows you exactly where your ledge climb fails. This should help speed up initial configuration.*

Reach and Grasp Decision

The following decision tree describes how the standard ledge climb determines if and when a reach and grasp should be triggered.



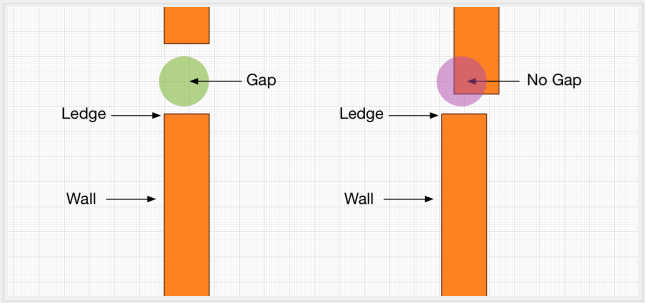


Decision Details

Decision	Responsible Component	Details
Wall Detected	Character	Uses the characters wall detection in combination with the side colliders to determine if the character is near a wall.
Required Colliders	Character	Checks that enough colliders are colliding. Available in editor but usually you don't need to adjust (default 2). If its too high you won't be able to grab thin platforms.
Ledge Detected	Movement	Determine if the wall we are hitting is a ledge. Handled by the Ledge Detection Method. Use NONE if you want to put all your ledges in a specific layer or give them all a tag. Use BOX_COLLIDER if your ledges are in the standard layer but are always use BoxCollider2D. Use CIRCLE_CAST to handle more complex geometry.
Within Reach Limits	Movement	Is the character the right y distance from the ledge. This is a min and max offset that specifies the distance between the ledge and the characters pivot. To set leeway use the *Grasp Leeway* setting found in the *Details* section of the ledge climb settings. You can also press the **Default** button which will estimate your characters reach point based on their overall height.
Gap Detected	Movement	Is there room for the character to stand up after the ledge climb (see image below). Use NONE if using a specific tag or layer for ledge climbing. Use RAYCAST_SIDES to use ray casts to check for room.
Input correct	Movement	Check that the user is pressing towards the ledge.
Reach Time Met	Movement	In order to provide smoother animations you can require the character to reach out for a ledge for a minimum amount of time before they can grab it. Note that this makes it harder for a player to grab the ledge so use with caution. To set leeway use the *Minimum Reach Time* setting found in the *Details* section of the ledge climb settings.

ledge climb settings.

Gap Detection

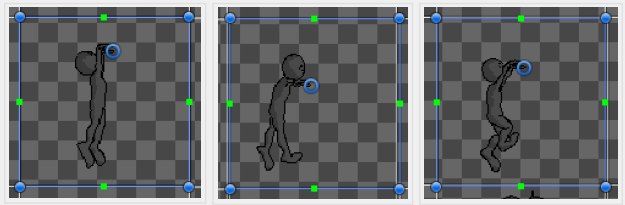


Sprite Set-up (2D)

To work with sprites set the **Animation Targeting** to **SPRITE_PIVOT**.

Each frame in the animation for the ledge hang and ledge climb animations should be set up such that the sprite pivot is the point that the character is hanging from (i.e. typically the hand).

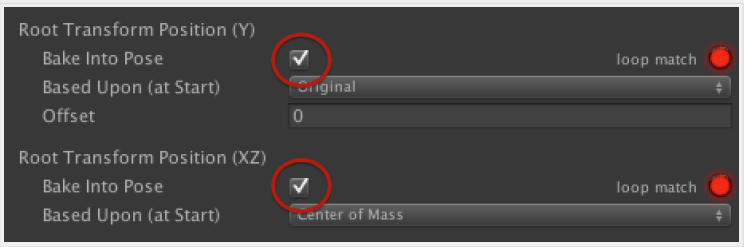
Sample sprites with pivots set up around the hand:



Model Set-up (3D)

To work with 3D models set the **Animation Targeting** to **BAKED**.

Similar to sprites the character animations should pivot around the ledge hand point. The animations should be set up such that the movement is baked in to the animation:

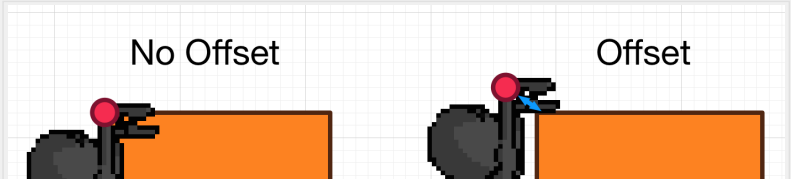


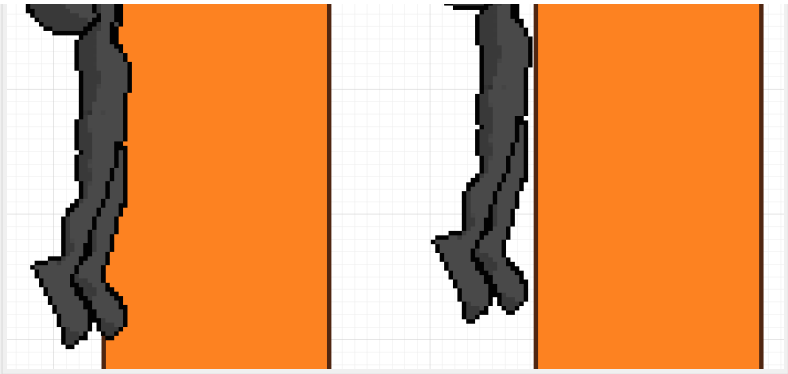
You would expect to see **red** *loop match* indicators as the X and Y positions will not be matched.

When you set the state to baked you need to pick which bone the character hangs from (called the *Grasping Bone* in the settings). Typically its one of the hand bones.

Ledge Offset

The ledge offset is the distance between the detected ledge hang point (for example the corner of the *BoxCollider2D*) and the characters pivot point in the ledge hang animation:



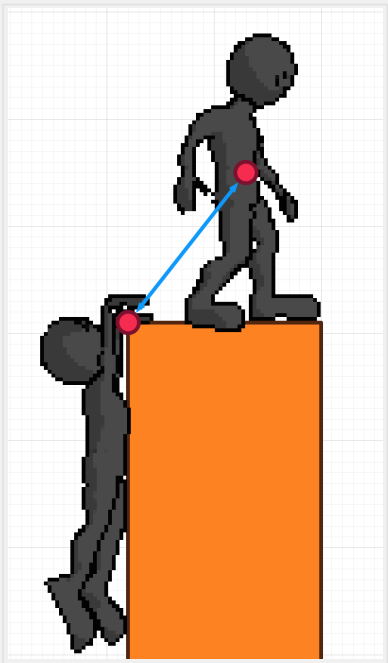


The **red** circle indicates the pivot. The small **blue** arrow indicates the offset, it is used to slightly tweak the hang position of the character. Typically this is more useful for 3D models or when you are using CIRCLE_CAST ledge detection.

Stand Offset

The stand offset is the difference between the pivot point of the final ledge climb frame and the first standing frame. For sprites this is also the same as the distance between the default pivot (for example sprites centre) and the pivot of the first ledge climb frame.

Sample stand offset for a sprite:



The **red** circles indicates the pivots. The **blue** arrow indicates the stand offset.

***Note**: if you are using gap detection the stand offset will be used as part of gap detection too (we need to check where the character will be standing after the offset is applied and make sure there is a gap).*

View page

– Wall Sliding

There are many different wall slides available in Platformer PRO. You can see some samples here:

https://www.youtube.com/watch?v=GeJ2yKehcGM

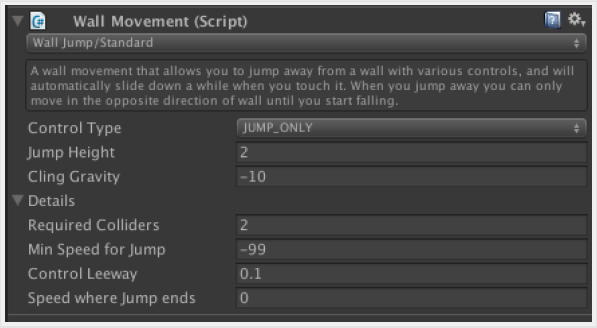
Basic Setup

The basic steps for setting up a wall slide are as follows:

- Add a WallMovement to your character GameObject or one of its children.

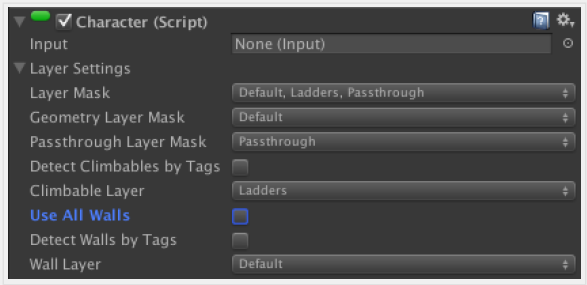
Remember: *Like all movements this is given priority based on its position so its best to move it high in the list so it takes priority over your default AirMovement.*

- Select one of the available WallJumps. In this sample we use about *Wall Jump/Standard*.



Remember*: You can hover over a fields label to see details of what it does.*

- Select your Character GameObject and open the layer settings



- Choose how your character detects walls, either sliding on all walls, or sliding only on walls with certain tags or in a certain layer.

Controls

The controls differ for each Wall slide but for most you press towards the wall to slide and the jump button to jump away from wall (wall jump). Some have different control options.

View page

Animation

– Animation Bridges (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO

An Animation Bridge is used to turn Character AnimationStates in to animations. This separation between Character and animation allows Platformer PRO to be used with any animation system.

Out of the box there are several AnimationBridge options but you can also write your own by implementing the very simple IAnimation interface.

Choosing an Animation Bridge

Your game has…	Then use…	Also consider…

Very simple 2D animation	Mecanim 2D	None.
2D animation	Mecanim 2D with transitions	Mecanim 2D
Complex 2D animation	Mecanim 2D with transitions	Mecanim 3D
Simple 3D animation	Mecanim 3D	Mecanim 2D with transitions
3D animation	Mecanim 3D	None.
Legacy 3D animation	Legacy	None.
Other	Your own custom bridge.	More bridges coming from JNA Mobile soon.

View page

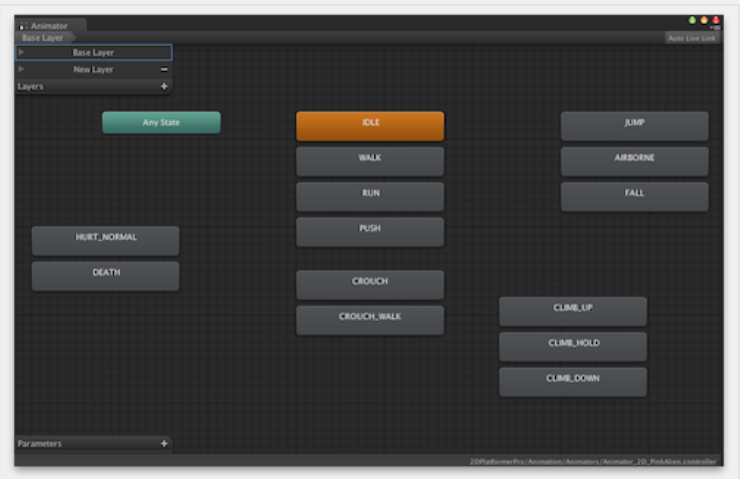
– Mecanim Animation Bridge - 2D

The **Mecanim 2D** animation bridge simple plays 2D animations when animation state changes.

It supports:

- Directly Playing animations without transitions.
- Simple animation priority (higher priority animations must play to completion before lower priority animations play)
- Animator Overrides

***Warning**: The Mecanim 2D animation bridge requires all states supported by your character have matching states in the Mecanim Animator (animation state tree). These names must be an exact upper-case match with the state name.*



Quick Start

This quick start assumes you have a character, and have already created animations for your sprite.

1. Create an Empty GameObject that is a child of your Character. Name it *CharacterSprite* or something else meaningful.
2. Drag your characters Idle animation to the newly created GameObject.

3. Select the prefab *Template-Mecanim2D-AnimationController* and duplicate it. Rename this duplicate something meaningful.
4. Assign this newly created prefab to the Animation Controller property of the Animator component of your newly created sprite GameObject.
5. Open the Animation view.
6. Drag your characters sprite animations to each matching animation state.

You should now have an animated character.

Required Components

In order for the Mecanim 2D bridge to animate it requires:

- An Animator component.
- A SpriteRenderer or MeshRenderer or SkinnedMeshRenderer (and associated components).

Configuration

The Mecanim 2D animation bridge relies on an Animation Controller with animation state names being the same the AnimationState enum. This means the state name should be an exact match with capital letters, for example: WALK or AIRBORNE.

View page

– Mecanim Animation Bridge - 2D With Transitions

The **Mecanim 2D with Transitions** animation bridge plays 2D animations when animation state changes and has the option to play *transition animations:* animations that are played when moving from one state to another.

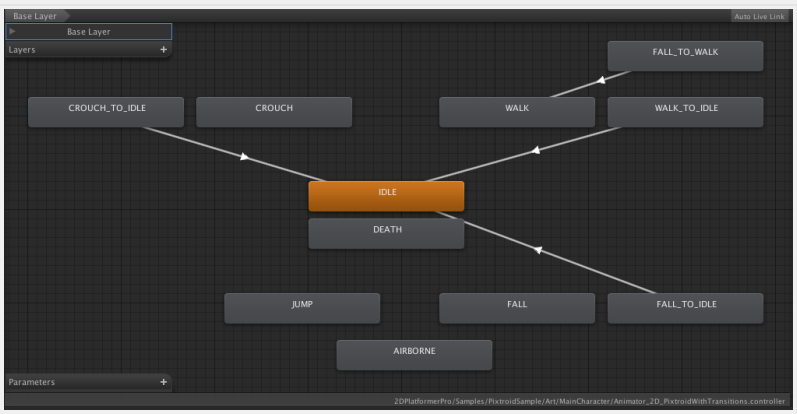
As an example you could create a landing animation by playing a new animation between the state FALL and the state IDLE.

It supports:

- Directly Playing animations without transitions.
- Simple animation priority (higher priority animations must play to completion before lower priority animations play)
- Animator Overrides
- Transition animations which play between one state and another.

***Warning**: The Mecanim 2D animation bridge with transitions requires all states supported by your character have matching states in the Mecanim Animator (animation state tree). These names must be an exact upper-case match with the state name.*

A sample Mecanim 2D with Transitions animation state machine, notice the states like CROUCH_TO_IDLE which includes a transition to the IDLE movement:



Quick Start

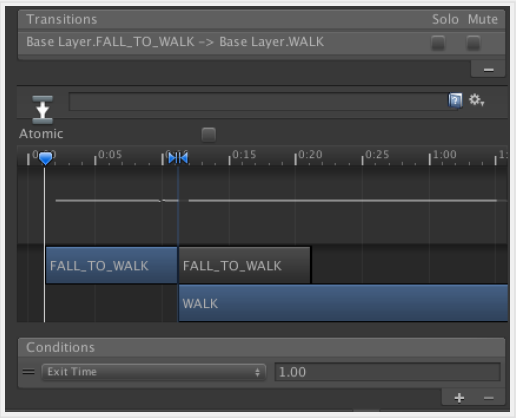
In the quick start we will set up a landing animation which occurs when character goes from the FALL state to the IDLE state or when the go from the FALL state to the WALK state.

***Note**: The steps assume you have a landing animation ready.*

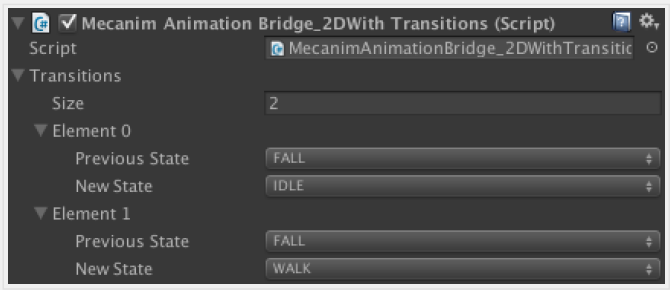
Steps:

1. Set up a 2D animator following the instructions from the Quick Start section of the 2D Bridge.
2. Add new states to your animator called: FALL_TO_WALK and FALL_TO_IDLE.
3. Add your landing animation to these states.
4. Add a transition from FALL_TO_WALK to WALK with an exit time of 1 (see screenshot below).
5. Add a transition from FALL_TO_IDLE to IDLE with an exit time of 1.
6. Go to your bridge and add 2 Transitions one for FALL to WALK and one for FALL to IDLE (see screenshot below).
7. Press play and you should now have a landing animation.

Transition with exit time 1:



Bridge with two Transitions:



Required Components

In order for the *Mecanim 2D with Transitions* animation bridge to animate it requires:

- An Animator component.
- A SpriteRenderer or MeshRenderer or SkinnedMeshRenderer (and associated components).

Configuration

The *Mecanim 2D with Transitions* animation bridge relies on an Animation Controller with animation state names being the same the AnimationState enum. This means the state name should be an exact match with capital letters, for example: WALK or AIRBORNE.

View page

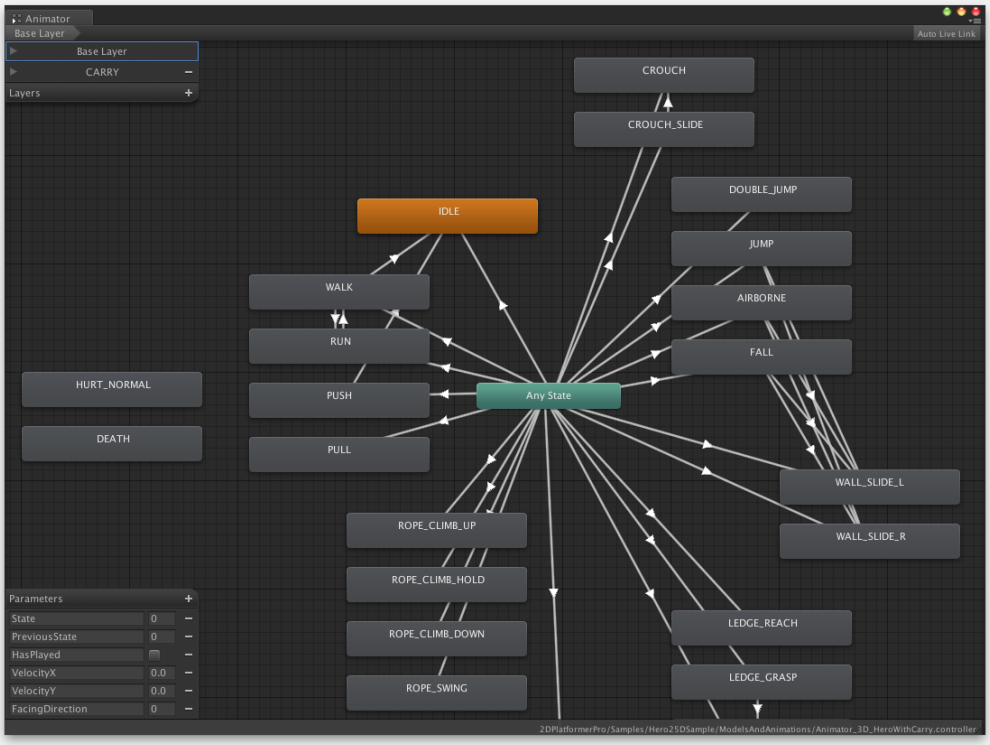
− Mecanim Animation Bridge - 3D

The **Mecanim 3D** animation bridge sets parameters in a standard mecanim state machine, it does not play states directly. You can use these parameters to drive transitions in the 'standard' mecanim manner.

***Tip**: Although the bridge is called 3D it can be used to do complex 2D animation too. For example see the Pixtroid sample scene*

Tip: Although the bridge is called 3D it can be used to do complex 2D animation too. For example see the Fixfold sample scene.

***Tip**: The Mecanim 3D animation bridge requires an understanding of mecanim. If you are starting out you may want to start with Unity's mecanim tutorials.*



Required Components

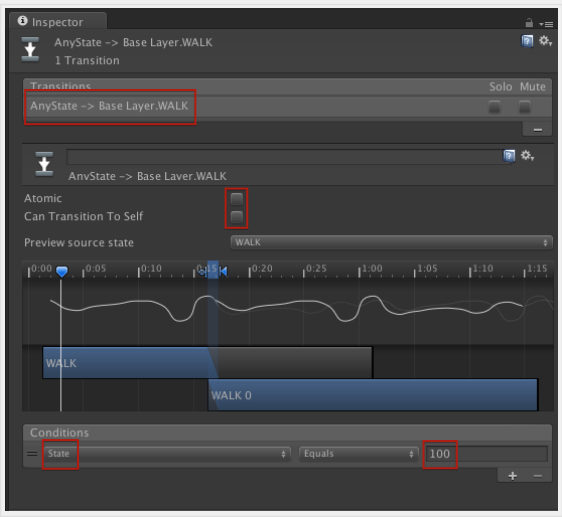
In order for the Mecanim 3D bridge to animate it requires:

- An Animator component.
- A SpriteRenderer or MeshRenderer or SkinnedMeshRenderer (and associated components).

Quick Start

The easiest way to create your 3D animations is to create transition from the *Any State* state to each animation state. You set the condition for this animation to the state number that matches the AnimationState.

For example the WALK state from the *AnimationState* enum has an integer value of 100. So you would set up walk with the following transition:



Can Transition To Self must be false (else the start of the animation would be repeated). Usually you would also set *Atomic* to false so that the animation can be interrupted.

You can find the integer value of an AnimationState enum by opening the .cs file *AnimationState:*

```
/// <summary>
/// Enumeration of all supported animation states.
/// </summary>
public enum AnimationState
{
    NONE            =       -1,

    IDLE            =       000,
    IDLE_ARMED      =       001,
    IDLE_ALT0       =       010,
    IDLE_ALT1       =       011,
    IDLE_ALT2       =       012,

    WALK            =       100,
    SLIDE           =       105,
    SLIDE_DIR_CHANGE=       106,
    RUN             =       110,
    CROUCH          =       120,
    CROUCH_WALK     =       130,
    ROLL            =       131,
    CROUCH_SLIDE    =       132,
```

***Tip**: It can help to have this open in a sticky window while setting up animation.*

State to State Transitions

In order to customise the transition between two animations states (such as the transition time) you need to do two things:

Step one - Create a transition between the two states

This is identical in set up to the transition between the *Any State* transition with a condition set on the State variable matching the integer value from the *AnimationState enum*.

Step two - Prevent the Any State transition from triggering

Let imagine two transitions:

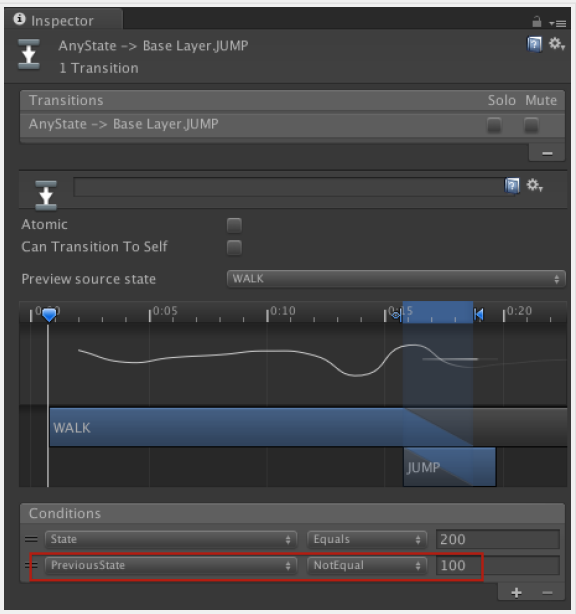
Any State to JUMP (the standard transition)

and

WALK to JUMP (the new transition we just added)

We now have two ways to get between WALK (which matches both WALK and *Any State*) and JUMP. So we need to prevent the *Any State* to JUMP transition if the previous state was WALK.

We can do this using an extra condition on the *Any State* transition which requires the PreviousState to be something other than WALK:



So now our *Any State* to JUMP transition only occurs when the current state is JUMP (200) and the previous state is **not** WALK (100).

Transitional Animations

You can add transitional animation in between state animations using the same technique as above. For example to add an animation for landing we need an animation between the FALL state and the IDLE state. To do this we do the following:

1. Create a new state called LAND.

2. Create a transition from FALL to LAND.

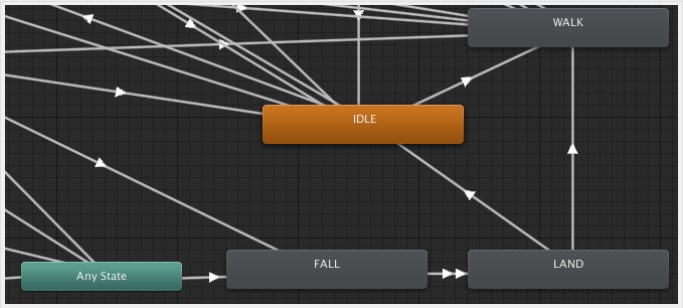
3. Set the condition on this transition to be: *State Equals 0* (IDLE).

4. Create a transition from LAND to IDLE.

5. Set the condition on this transition to be the exit time (we want the landing animation to finish then to walk as normal).

6. Finally guard the *Any State* to IDLE transition by adding a new condition (as per above) so that it doesn't play if the PreviousState was FALL. The condition would be: *PreviousState NotEqual 220* (FALL)

***Tip**: You would probably want to add additional transitions from LAND to other states like WALK or RUN.*



Blend Trees

Blend trees can be set-up as per standard mecanim animation practices. The Stealth Sample has a simple blend tree between WALK and RUN based on the setting of the *VelocityX* parameter.

Available Parameters

The following parameters are made available to the mecanim animator:

State	Animation state last sent by the character.
PreviousState	Animation state that was in effect prior to the current state.
HasPlayed	Tracks if the current animation has played for at least one frame (this is an advanced option, see details below).
GunPositionX	X value of the aiming direction. Only set if a ProjectileAimer is attached to the character.
GunPositionY	Y value of the aiming direction. Only set if a ProjectileAimer is attached to the character.
VelocityX	Character velocity in X. As set by the movement, remember that movements can interpret velocity in different ways (relative, world, or something custom).
VelocityY	Character velocity in Y. As set by the movement, remember that movements can interpret velocity in different ways (relative, world, or something custom).
FacingDirection	Int value for facing direction 1 = right, 0 = none, -1 = left.

Exposing new Parameters

With a little code you can easily extend the Mecanim 3D bridge to expose new parameters to help improve your animations. This code sample below creates a new bridge which exposes character health and slope rotation as new parameters for your mecanim animator.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

namespace PlatformerPro
{
    /// <summary>
    /// An animator that sends more parameters to the mecanim controller than the standard bridge. This is primarily
    /// here as an example of how you can extend the Mecanim 3D bridge.
    /// </summary>
    public class MecanimAnimationBridge_3DExtraParams : MecanimAnimationBridge_3D
    {
        protected CharacterHealth characterHealth;

        /// <summary>
        /// Init this instance.
        /// </summary>
        override protected void Init()
        {
            base.Init ();
            if (myCharacter is Character)
            {
                characterHealth = ((Character)myCharacter).GetComponentInChildren<CharacterHealth>();
            }
        }

        /// <summary>
        /// Unity Update hook.
        /// </summary>
        override protected void ActualUpdate()
        {
            base.ActualUpdate ();
            myAnimator.SetFloat("SlopeRotation", myCharacter.SlopeActualRotation);
            if (characterHealth != null)
            {
                // Percentage health we could for example use this to blend in limping animations as the character takes damage.
                myAnimator.SetFloat("Health", characterHealth.CurrentHealthAsPercentage);
            }
        }

    }
}
```

View page

– Direction Facers

A Direction Facer controls how your character handles a change in the characters FacingDirection property. It is used to ensure your character faces the right direction.

Out of the box there are several Direction Facersnoptions but you can also write your own.

Direction Facers

Type	Description	Documentation Link
UnitySpriteDirectionFacer	Responds to direction changes by inverting the x scale of the GameObject it is attached to. Usually this would be a sprite renderer where inverting the x scale mirrors	

	the character.	
UnitySpriteAimDirectionFacer	Similar to the UnitySpriteDirectinoFacer but instead of using the facing direction it uses the aiming direction. With this you can have a character who faces one way while moving the other.	
ModelRotationDirectionFacer	Responds to direction changes by rotating the model around the Y axis. Generally used for 3D models. Not for use with animations which apply root motion (Advanced).	
LeftRightAnimatorOverrideDirectionFacer	v1.1.0 Only Uses a different set of animations for each direction via the use of an AnimatorOverrideController. Typically used when you have a character who is not visually symmetrical (for example see Bionic Cop with his Bionic left arm).	Left/Right Direction Facer

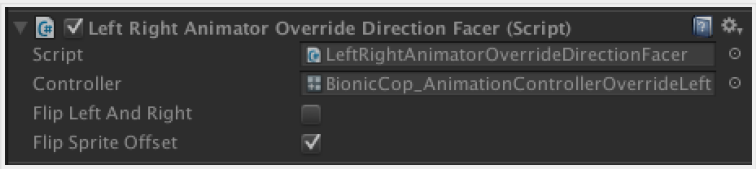
View page

– Left/Right Direction Facer

The Left Right direction facer (LeftRightAnimatorOverrideDirectionFacer) uses a different set of animations for each direction via the use of an AnimatorOverrideController.

Typically used when you have a character who is not visually symmetrical (for example see Bionic Cop with his Bionic left arm).

Parameters

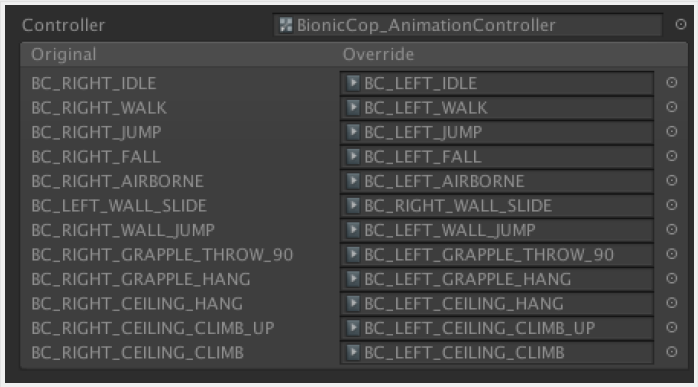


Controller	Link to the AnimatorOverrideController to use for the alternate directions
Flip Left and Right	If false then the right facing animations are the default and the left facing animations are the override. If true its the other way around.
Flip Sprite Offset	If true then the x offset on this GameObject will be flipped when facing direction changes. Usually true.

Setting up An Animator Controller Override

Because Unity's documentation is not very detailed we elaborate the set up here.

1. Start by creating a normal character animator controller with all the animations for a given direction (usually right).
2. Use the asset menu (**Asset -> Create -> Animator Override Controller**) to create a new override controller.
3. Assign your normal Animator Controller to the Controller field of the new Animator Override Controller.
4. You should notice all the animation states form your original controller appear in the list. Add the substitution animation will be used when the character is facing in the opposite direction.



***Note**: You can leave some of the animation states blank if you don't wish to override them.*

***Warning**: Don't use the same animation for more than one state in your original animator controller. Instead duplicate it and rename. For example if you use an animation called AirAnimation for both AIRBORNE and FALL states, you should instead use two copies of the same animation by duplicating and renaming.*

***Warning**: Some Unity versions have problems with animator override controllers, avoid using Unity 5.1.0 through to 5.2.2.*

View page

Health and Damage

− Character Health (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO.

The *CharacterHealth* component is responsible for tracking and handling any damage to the character. This includes both health and lives as well as the responses to health related events like character death or game over.

Most characters will have a CharacterHeath component. To use add a CharacterHealth component to the same GameObject as your character.

***Note**: To take damage a character will also requires a CharacterHurtBox. Without a CharacterHurtBox a character can only be sent the Kill() message which triggers death but not damage.*

Properties

CharacterHealth has the following properties:

Starting Health	The amount of health the character starts with.
Max Health	The maximum amount of health the character can have.
Starting Lives	The number of lives the character starts with.

Max Lives	The maximum amount of lives the character can have.
Invulnerable Time	The time in seconds that the character is invulnerable for after they take damage.
Death Actions	A list of actions to do when the character dies (for example reload the scene).
Game Over Actions	A list of actions to do when the game is over dies (for example reload the menu screen). Note if your game uses a GameOver Screen then you may want to defer some (or all) Game over Actions and let the Game Over Screen handle them.
Skip Damage animations on Death	If true then the damage animation will not play when the character dies (instead just the death animation will play).
Skip Death actions on Game Over	If true then the death actions will not fire when the character loses their last life.

Death and Game Over Actions

The following actions can be triggered when the character dies:

RESPAWN	Respawn the character at the given respawn point or the last registered respawn point if none supplied.
RELOAD_SCENE	Reload the current scene.
LOAD_ANOTHER_SCENE	Load a different scene as specified in the Supporting Data.
SEND_MESSAGE	Sends the Supprorting Data as a message go the Supporting GameObject. Generally you shouldn't need to do this, for more customised behaviour see events below.
CLEAR_RESPAWN_POINTS	Clear all currently saved respawn points.
DESTORY_CHARACTER	Destroy the character GameObject. Sometimes useful if you have multiple character objects and want to load them without reloading scene.
RESET_DATA	Reset all Persitable components attached to the Supporting GameObject.
RESET_SCORE	Reset the score with an id defined in Supporting Data.

See events below if you wish to perform other actions on death.

Character Health Events

Like most components in Platformer PRO CharacterHealth sends a number of events which you can use to attach related behaviour (for example play a sound and particle system when the character is damaged).

CharacterHealth sends the following events:

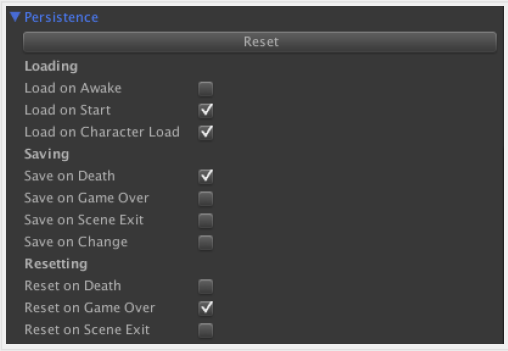
Healed	Sent when the character gains health. Includes details of how much was healed.
Damaged	Sent when the character is damaged. Includes details of the damage amount, damage type and the damage direction.

Died	Sent when the character dies. Includes details of the damage that caused the death (as per Damaged event).
GameOver	Sent when the game ends (i.e character dies with 0 lives). Includes details of the damage that caused the death (as per Damaged event).

Persistence

CharacterHealth is a *Persistable* so you can for example store the characters health between levels. Both current health and current lives are saved.

Typical settings for a character:



This effectively ignores health because only death causes a save. If you also want to save current health add check to the checkbox for save on scene exit.

Note that the CharacterHeath does not have an implementation for save on change. If you wish to save health data after every change, you could extend the CharacterHealth class and add a change handler to the damage methods, but this is generally not necessary.

View page

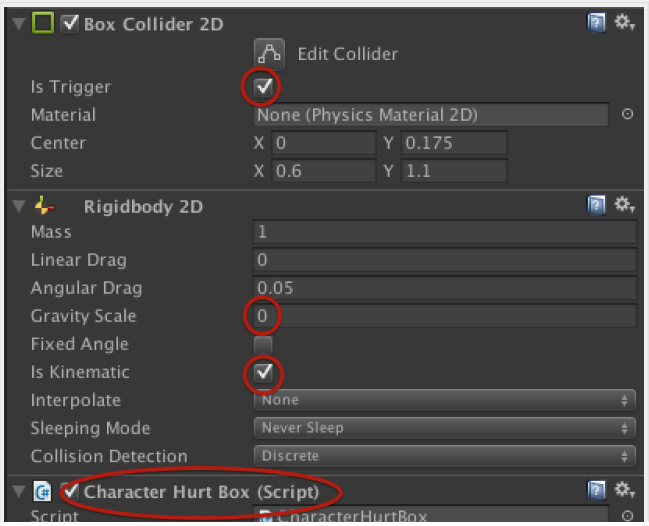
− Character Hurt Box

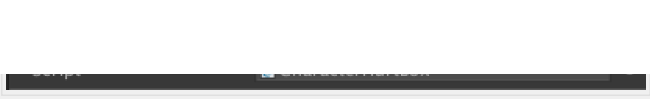
A character hurt box is a 2D trigger (and collider) used to register damage. Without a *CharacterHurtBox* a character can only be sent the Kill() message which triggers death but not damage.

The CharacterHurtBox component is typically added to a child GameObject of the characters model or sprite. This way it will flip when the model or sprite is flipped.

This component also requires some kind of *Collider2D* (for example a *BoxCollider2D*) and because it moves should also include a *Rigidbody2D*.

The Collider2D **must** be a trigger, the Rigidbody2D **must** be kinematic with a gravity scale of 0:





Layers

The default layer for a Character Hurt Box is the *Character* layer.

If you are deviating from the standard layers then make sure:

- Your character does not collide with the hurt boxes layer (i.e. its not a Geometry or Passthrough layer)
- The layer collides with the hazard/enemy projectiles layer (hitting an enemy or hazard is what triggers the damage).

***Tip**: If you have a HurtBox set up you often don't need any other colliders as you can also use the hurt box to collect items and perform other interactions.*

View page

Items and Power Ups

– Items

Platformer PRO provides support for various kind of collectible items.

To get started with Items:

1. Add an *ItemManager* to your Character GameObject (see ItemManager below).
2. Make sure your character has a correctly set up HurtBox or other CharacterReference.
3. Add a GameObject to represent your item. Make sure this is in a non-geometry layer that can interact with Characters (e.g. the '*Collectible_Projectile*' layer that is setup by default).
4. Add the *Item* component to your Item GameObject.
5. Add a 2D collider to your Item GameObject.
6. Create a GameObject that is a child of your Item GameObject
7. Add a visible component (such as a *SpriteRenderer*) to this child GameObject.

***Note**: Its not vital that the visible component be a child but this is good practice as it allows you to effect the visible component of the item without affecting its behaviour.*

Item Classes

There are four types of items:

STACKABLE	An item that can be stacked up in the inventory until some maximum is reached. For example coins or stars.
SINGLE	An item of which the player can only have one.
KEY	Special item type used to unlock doors. See Doors for more information.
POWER_UP	Special item type which grants the user new powers. See Power Ups.

Item Type

The item type is a string identifier for the type of item. It can be anything you want, but remember items with the same ID should, for all intents and purposes, be the same.

Create a Stack of Items

If you wish to add more items to a stack (e.g. 10 bullets instead of 1), update the Item component to instead be a StackableItem. And enter an amount in the Amount field.

Item Manager

The ItemManager component handles collecting and using items. It **must** be added to your Character GameObject or to a child of your Character GameObject.

Configuring Stacks of Items

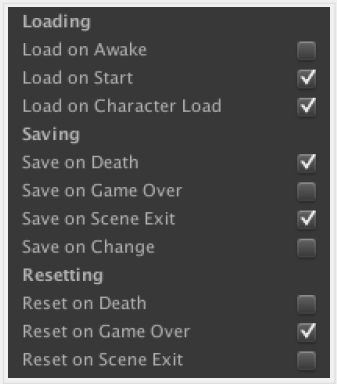
The ItemManager allows you to configure stackable items by adding one or more items with the following details:

Type	Unique ID for the item type (e.g 'fuel' or 'COIN').
Max	Maximum number of items allowed (i.e. how many can the Character hold).
Starting Count	Allow your character to start with some of this item. For example you may want to start with 20 bullets or 100 fuel.

Configuring Persistence

The ItemManager is a *Persistable* (see Persistables) and thus allows you to persist the state of the items between levels, games or even play sessions.

Typically you want to load Item data when the level starts or the Character loads and save it when the level ends or the character dies. When the character loses all their lives (GameOver) you would usually reset the data.



***Warning**: Be mindful of the effects of saving data. For example if your level reloads on death and your ItemManager does not reset on death the character will be able to collect the same objects multiple times. This is fine for some games, but not for others.*

View page

− Power Ups

Power Ups are a special type of item which grant your character new powers. Power-ups may last for a time-limit, or alternatively last until a level is complete (or the character dies).

Power Ups are created in the same ways as *Items* but instead of requiring an ItemManager they require a *PowerUpResponder*. The PowerUpResponder defines what happens when you collect a power-up.

***Note**: Power-ups are not permanent. A future version will contain **Upgrades** which are more permanent variation of power-ups (last between levels, last between lives or even last between games). Let us know if you want this feature prioritised.*

Power Up Responder

The *PowerUpResponder* component needs to be added to your Character in order for Power Ups to have an effect on the Character.

The PowerUpResponder defines a list of power ups and the actions that need to be taken to enable the power. It also defines the **reset** actions: those things that need to be done to undo or reset a power up.

Defining a Power Up

Click the **Add Power Up Type** button to add a power up. A new power up type with an empty name will be added to the list. Expanding the empty item will show the following fields:

Power Up Type	Name of the power up. Must match the item type exactly.
Power Up Timer	If the power up expires after a time then this is the time in seconds that the power up will last for.
Reset Response	Only shown if Power Up Timer is non zero. This is the 'reset' power up that will be used to reset the character back to their default state when the timer expires.

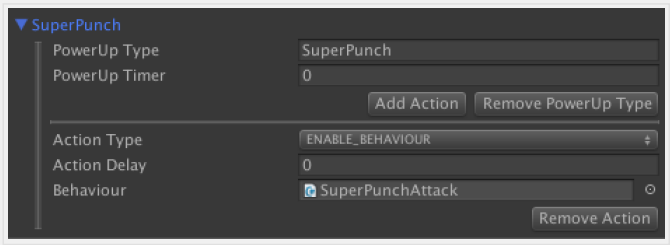
To remove a power up click the **Remove Power Up Type** button.

Power Up Actions

Each power up type should have one or more actions that implement the power up behaviour. To add an action click the **Add Action** button.

Power up actions are identical to event responses, see Event Responder for full details.

As an example the following PowerUp is called SuperPunch and enables a BasicAttack behaviour:

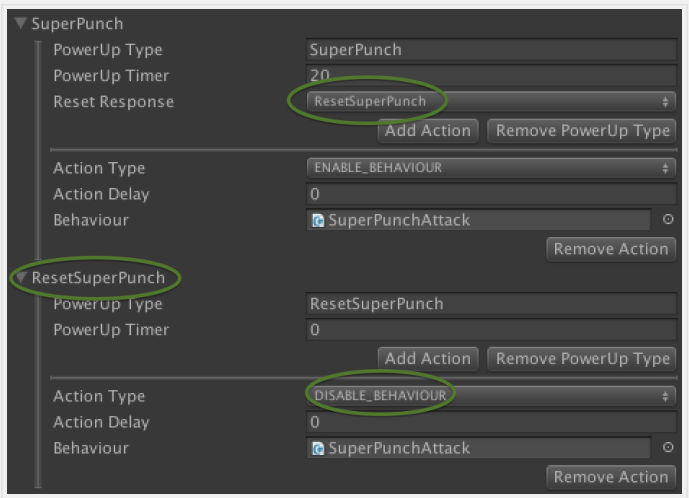


To remove an action click the **Remove Action** button.

Defining a Reset

A reset is exactly like any other power up except that it defines actions that undo the actions of another power up. Resets are required for any timer based power up.

Here the example of the SuperPunch is extended to include a 20 second timer and a reset which disables the SuperPunch:



The Master Reset (RESET)

The Power up called RESET is known as the *master reset*. It is used to reset your Character after death. If your Character set up so that the you do not RELOAD_SCENE after death then you need to ensure your master reset has all the actions required to reset any and all power-ups.

For example if you have one power-up that makes your character invulnerable, and another which changes the animation override, and another which enables an ability, then the master reset should undo all three actions (i.e. make vulnerable, reset the animation override, disable the ability).

***Note**: If your character is set to RELOAD_SCENE as part of the death actions you do not need to specify a Master Reset (as reloading the scene will reset the Character and Scene back to default).*

View page

Enemies

Enemy Overview

LIMITED ENEMY OPTIONS ARE AVAILABLE IN THE LIGHT EDITION

Use the *Enemy* component to create your bad guys. Enemies cause damage to players, but unlike basic Hazards they can also perform actions like moving around, shooting, hiding, and so on.

This section has the following sub-sections:

Enemy Properties	Describes enemy properties which are a simplified set of properties similar to Character and CharacterHealth properties..
Enemy Design	Discusses various options for designing you enemy, such as how it moves and the optional AI component.
Hazards, HitBoxes and HurtBoxes	Describes how enemies cause and take damage.

View page

Enemy Properties

This article lists the properties of an Enemy.

Enemy Health

Enemy health has the following properties:

Health	Enemies have a starting health which you can adjust using the slider or text field.
Invulnerability Time	How long should the enemy be invulnerable to damage after being hit (measured in seconds).

Enemy Interactions

These settings control general interactions between the Enemy and other components such as Enemy Movements and Platforms.

Continue Movement On Fall	If false the character will set the AnimationState to falling when the character falls and no Movement will control the character. If true movement will still be able to control the character in the falling state.
Switch Colliders On Direction Change	If true the left and right colliders will be flipped when the character changes direction. This is primarily for asymmetric characters.
Start Facing Direction	To easily align with your sprites you can set the initial facing direction.
Enemy Interacts With Platforms	If true this Enemy will be able to trigger and parent to platforms much like a Character. Note that not all platform types will work with enemies.

Enemy Collisions

Enemies use a much simplified version of the characters geometry collision system. Remember that not all enemies need geometry collisions, for example if an enemy simple walks back and forth on solid indestructible ground then there's no reason for it to check the ground beneath it.

To set up enemy collisions use the following properties:

Geometry Layer Mask	A list of layers that the enemy will bump into.

Geometry Layer Mask	A list of layers that the enemy collides with.
Can Character Fall	If true the character will need feet colliders and will fall if its feet colliders aren't touching something.
Terminal Velocity	The maximum speed the character can fall at (negative number).
Grounded Lookahead	How far above the ground can the enemy be and still be considered grounded. For enemies you can often leave this at 0, but if your enemy needs to snap to ground on slopes or be able to stand on vertically moving platforms then this will need to be increased.
Use Character Gravity	If true then the gravity applied to the enemy will be the same as the gravity applied to the character. The way the character is found is quite simple (FindObjectOfType) so this may not be suitable for multiplayer games.
Custom Gravity	If not using character gravity set the custom gravity here (usually a negative number).
Side Collisions	Does the enemy have side colliders. If not they will be able to walk through geometry that gets in their way.
Switch on Side Hit	If the character runs in to something should it turn around? Make sure your EnemyMovement supports this (most of the out-of-the-box EnemyMovements do).

Once your properties are configured you can use the **Edit Feet** and **Edit Sides** buttons to adjust Enemy colliders. Doing this will bring up movable handles in the in the Scene View which allow you to adjust the location of the feet and side colliders respectively.

Help! I Need more Colliders

To minimise performance impact of Enemies they perform a limited number of collisions (2 feet, 2 side, no head colliders).

***Note**: Side collisions are skipped if the enemy is moving in the opposite direction to the colliders direction. Generally this is okay, but if you have fast moving platforms that can push your Enemy you may need to customise.*

If you need more collisions you can extend the Enemy class or use a Character class for your Enemy. See also Enemy Design.

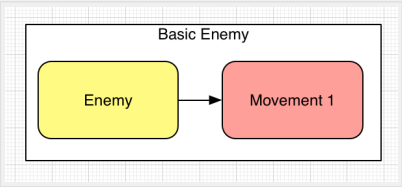
View page

– Enemy Design

Once your properties are configured you have a choice about how your enemy will be designed. There are two patterns you can use for your enemies: the **Basic Enemy** pattern and the **AI Enemy** pattern.

***Note**: you can also build very complex enemies as an extension of the Character class but that is outside the scope of this discussion. Drop us a support request to find out more.*

Basic Enemy



A basic enemy pattern uses two controlling components:

- the *Enemy* component which contains details of the enemy such as its health and collision boundaries,
- the *Movement* component which is a subclass of the *EnemyMovement* class and controls how the character moves.

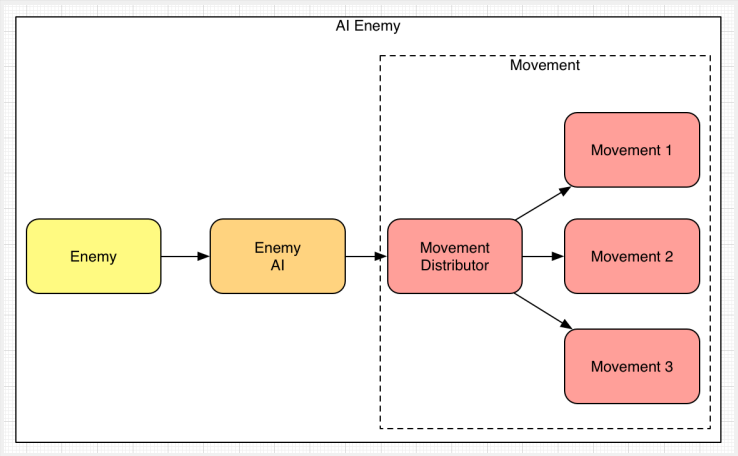
Each frame the Enemy object passes control to them movement in a similar manner to a Character movement.

You may optionally add an *EnemyDeathMovement* which will take over movement control when the enemy is damaged or when the enemy dies.

This pattern allows you to quickly create enemies with simple behaviour like patrolling back and forth. You may also use this pattern to create more complex behaviour by extending the EnemyMovement class with your own implementation.

The drawback of this approach for complex enemies is that the most of the behaviour is embedded in the movement class, making it difficult to share only parts of the behaviour between enemies. For example you might want many enemies to detect and shoot at the character in a similar way, even though they move in many different ways (fly, run, walk, swim, etc).

AI Enemy



Instead of having one movement the AI Enemy has a *MovementDistributor* which associates different enemy states with many different movements. The enemies state is decided by an *EnemyAI* component which picks an enemy state based on its own logic (for example an enemy AI might change to the shooting state when a Character is close by).

This pattern allows you to create reusable AI components and reusable movement components which you can combine to create new enemies.

The drawback of this approach is that it can be more complex to configure and debug your enemies. The following diagram might help you better understand the control flow of an AI Enemy:

<picture>

***Tip**: The Enemy class also provides a debugging view which can help with this.*

Choosing an Enemy Pattern

Most of the sample use the AI Enemy pattern so that their component parts can be easily reused in your own games but this does not mean you need use this pattern.

There are no fixed rules but here are some dot points:

- If your enemies all have very distinct movements with very little shared behaviours you should consider the Basic Enemy pattern. You can still paramatize aspects of the movement to allow for some variation.
- If your enemies sense the character you should consider the AI Enemy pattern as there are examples of sensing the character already available.
- If your enemies often have some shared movements (for example they all shoot in a similar way) but other different movements (some fly, some walk, etc) you should consider the AI Enemy pattern.

View page

The Attack System

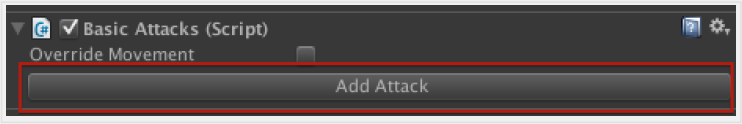
− Attack System Overview

The attack system allows you to add melee and projectile attacks to your character.

To get started add the *BasicAttack* component to your Character game object.

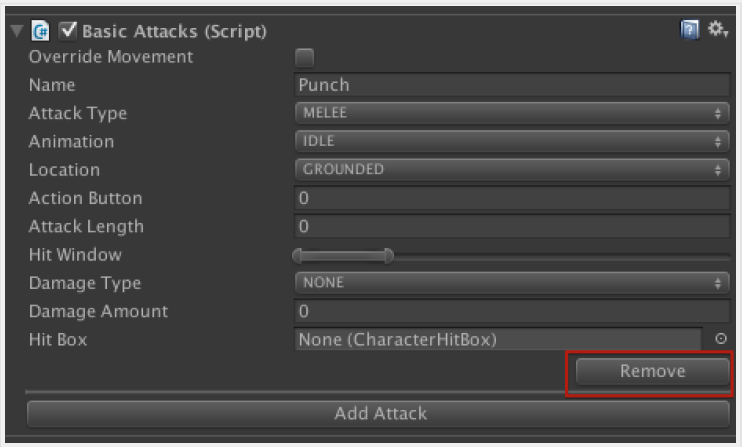
Attacks

The Basic Attack component can define multiple attacks. To add a new attack press the **AddAttack** button:



You can add more attacks by pressing this button again.

You can remove attacks by pressing the **Remove** button below the attacks details:



Next Steps

To read about Projectile attacks see:

To read about Melee attacks see:

View page

– Projectile Attacks

Projectile attacks fire a projectile prefab when a key is pressed.

Creating a Projectile Prefab

In order to fire projectile we need a prefab that represents the projectile.

***Tip**: The instructions below cover every step but if you are experience with Unity you may shortcut the process by duplicating an existing projectile such as the 'CommandBro-Bullet' prefab.*

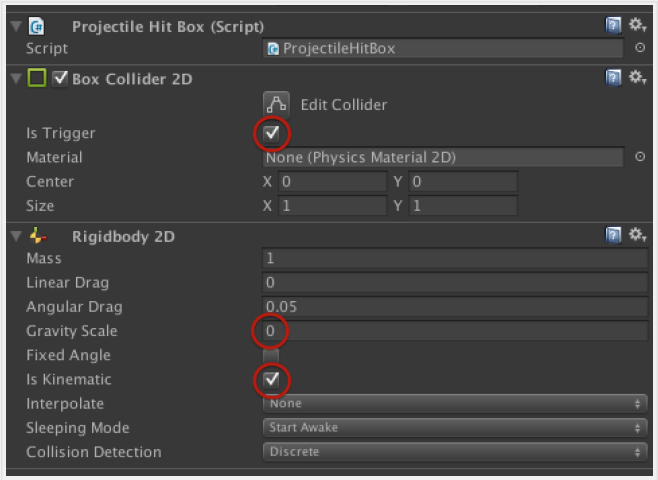
To create a projectile prefab:

1. Create an empty GameObject we will call this the *Projectile GameObject*.
2. Rename this GameObject to something meaningful. For example 'ProjectilePrefab'.
3. Put the Projectile GameObject in a layer that hits enemies. The default is the layer called 'CharacterProjectile'.
4. Add a *Projectile* component to this GameObject.

Next we create the HitBox which is the component that collides with enemies:

1. Add an empty GameObject as a child of the Projectile GameObject. We will call this the *HitBox GameObject*.
2. Rename this GameObject to something meaningful. For example 'ProjectileHitBox'.
3. Add a *ProjectileHitBox* component to the HitBox GameObject.
4. Add a 2D collider such as a *BoxCollider2D* to the HitBox GameObject.
5. Make the 2D collider a trigger.
6. Add a *Rigidbody2D* component to the Hitbox GameObject.

7. Make the Rigidbody2D *kinematic*.
8. Set the *Gravity Scale* to 0.



This projectile does not have a visible component. You probably want to add one. For example to add a sprite:

1. Add an empty GameObject as a child of the Projectile GameObject. We will call this the *Sprite GameObject*.
2. Rename this GameObject to something meaningful. For example 'VisibleComponent'.
3. Add a SpriteRenderer to the Sprite GameObject.
4. Use the Sprite field to pick a sprite for the projectile.
5. Change the layer of the sprite so its in the foreground or character layer (we want to draw projectiles on top of the scenery).

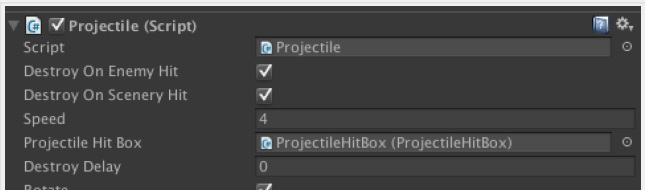
Your GameObject should look something like this:

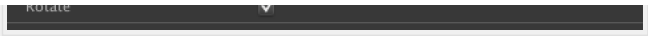
Projectile Settings

With the structure created we will now set up the projectile. The following settings are available:

Destroy on Enemy Hit	1. Should we destroy this projectile when we hit an enemy? If not checked the projectile will pass through enemies but will still cause damage. You can use this to create strong projectiles that damage multiple enemies.
Destroy on Scenery hit	Should we destroy this projectile when we hit the scenery? If not the projectile will pass through scenery even when on a colliding layer.
Speed	How fast does the projectile move?
Projectile Hit Box	HitBox for this projectile. Drag a reference to the Projectile HitBox you created above.
Destroy Delay	How long in seconds between triggering destroy and deactivating this GameObject. Typically used to give time for particle effects and sounds to play.
Rotate	Should we rotate projectile to match direction it is fired in.

Sample configuration:





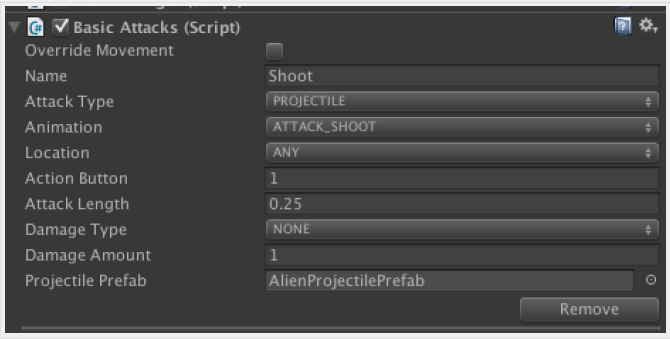
Create Prefab

With your settings done, drag the GameObject to the project window to create a prefab. Once the prefab is created you can remove the projectile from your screen.

Configuring a Projectile Attack

Now that we have created a projectile prefab we can configure our projectile attack. Ensure you have added a *BasicAttack* movement to your character GameObject then:

1. Add a new Attack using the **Add Attack** button.
2. Give the attack a name. For example 'shoot'.
3. Set the *Attack Type* to **Projectile**.
4. Choose an *Animation State* for the attack. For example **ATTACK_SHOOT**.
5. Set an *Attack Length* of **0.25**.
6. Set a *Damage Amount* of **1**.
7. Select your newly created projectile prefab as the *Projectile Prefab*.



Settings in Detail

The following settings are available for a projectile attack:

Override Movement	Should this attack take complete control of movement. For projectile attacks this is typically false.
Set Animation State	If true the attack system will set the animation state, ignoring the animation state from the active movement. If false an Animation Override will be set instead.
Name	A name for the attack.
Attack Type	PROJECTILE or MELEE. Use PROJECTILE for projectile attacks.
Animation	Animation State to set if *Override Movement* is *true*. Animation Override to set if *Override Movement* is *false*.
Location	Can this attack start on the GROUND in the AIR or anywhere (ANY).
Action Button	Action button to press to trigger this attack. Corresponds to index of action buttons set in the Input Configuration.
Attack Length	How long does the attack last. For PROJECTILE attacks this is the time between each shot.
Damage Type	What type of damage does this attack do. You can use NONE if it does a generic kind of damage or if you aren't using damage types.
Damage Amount	How much damage does this projectile do?

Projectile Prefab	Prefab to create when the attack is started.
Ammo Type	Set to a string matching the item type of the ammo you want to use. If empty attack will have infinite ammo.

Animating Projectile Attacks

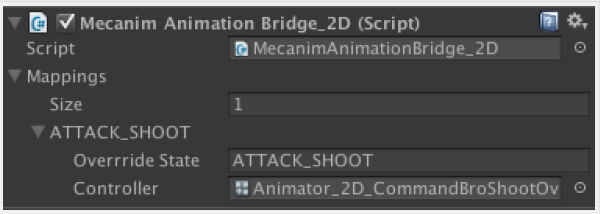
There are several options for animating projectile attacks.

Use Animation Overrides

For simple sprite based attacks you may wish to use an animation override which contains overridden animations that match the default sprites but indicate some kind of firing action. This is seen in the CommandBro sample.

To set this up:

1. Set *Override Movement* to **false**.
2. Set *Set Animation State* to **false**.
3. Create an *AnimatorOverrideController* (http://docs.unity3d.com/ScriptReference/AnimatorOverrideController.html) which maps shooting animations to the equivalent states.
4. Add an override mapping to your animation bridge and associate your attack animation state to the *AnimatorOverrideController* you created.



Set Animation State Without Control

With this set-up we animate the attack by setting the *AnimationState* just like any other Movement would BUT we do not take control of the characters position. The character position is still controlled by other movements.

This option can be quite effective way to allow a complex range of animation options but it typically works best if your attack animations are quite short.

Tip: Although normal movement is retained you can use the block movement settings to stop certain movements triggering during an attack (for example you may not want to be able to Jump or Climb in the middle of a punch attack).

To set this up create your animations just like any other state (see Animations).

Control Movements

With this set-up the attack system takes complete control of the movements. And all other movement will halt.

To set this up create your animations just like any other state (see Animations).

***Tip:** This BasicAttacks works okay for controlling grounded attacks, but more complex attacks will likely require specialised movements which extend the BasicAttack. For example see DashAttack.*

View page

Platforms and Geometry

– Platforms (I)

(I) IMPORTANT: Understanding this section is vital to using Platformer PRO

Plaforms are used for level geometry that needs to do something more than or different to standard level geometry. Examples include: platforms that move back-and-forth, trampolines, and slippery ice.

Friction

The simplest thing you can do with a platform is change the friction. This can make the platform seem sticky or slippery to the player. The following table lists some friction values you might consider, however note that the affect of friction also depends on your ground movement settings):

Friction Setting	Description
-1	Use characters default friction.
0	No friction.
1.5	Slippery Platform
2	Standard platform.
3	Sticky platform.

***Warning**: Not all ground movements apply friction (e.g. Digital/Basic).*

Activation/Deactivation

Platform is equipped with a boolean variable called Activated which is typically used to turn the effects of a platform on or off. A typical example would be an elevator which moves up and down when on, and does not move when off.

You can also configure the activation and deactivation of a platform to be automatic using the Automatic Activation and Automatic Deactivation fields. The following values are available.

Activation Types

NONE	No automatic activation.
ACTIVATE_ON_START	Activate when the level starts.
ACTIVATE_ON_STAND	Activate when the character stands on the platform.
ACTIVATE_ON_LEAVE	Activate when the character leaves the platform.

Deactivation Types

NONE	No automatic deactivation.
DEACTIVATE_ON_STAND	Deactivate when the character stands on the platform.
DEACTIVATE_ON_LEAVE	Deactivate when the character leaves the platform.
DEACTIVATE_ON_EXTENT	Deactivate when the platform reaches the end of its movement, for example when an elevator reaches its top or bottom position.

Parenting

When a character stands or in other ways collides with a platform you can choose to parent the character to the platform. You use parenting when you want the character to move with the platform, without relying on gravity and collisions. Most moving platforms should use parenting as it gives the smoothest movement.

To parent the character to a platform the platform must return 'true' to the Collide call described below (see Collisions).

***Warning**: Do not parent a character to a platform that is or can be rotated unless it has a uniform scale like (1,1,1). See FAQ for details on how to set up*

oddly shaped rotating, parenting platforms.

Built-in Platform Types

Platform Type	Parenting?	Description
BackAndForthPlatform	true	A platform that moves left and right.
DepthPlatform	false	A platform which only hits characters at a certain depth (see Depth).
DepthSwitchPlatform	false	A platform which changes the depth of the character as the walk through it (see Depth).
DestructiblePlatform	false	A platform which takes damage and is destroyed after too much damage.
Door	false	A special platform type for doors (see Doors).
FallingPlatform	false	A platform that falls after it has been stood on for a certain amount of time.
LoopPlatform	false	A platform used for creating a 360 degree loop.
Platform	false	Default platform, you can use this for friction control (e.g. ice).
SpringBoardPlatform	false	A platform that bounces the character in to the air when the y jump on it.
SpawnOnHeadbutt	false	A platform that spawns items when head butted. Similar to the mario [?] boxes which spawn coins.
SpecialMovementPlatform	false	Triggers a special movementGenerally used for acrobatic mecanim movements.
StompOnHead	false	A special type of platform which is attached to Enemies (see enemies) and allows you to damage or kill them when you jump on their head.
TriggerPlatform	false	A platform that behaves like a Trigger (see Triggers).
UpAndDownPlatform	true	A platform that moves up and down.

Note: This list is not comprehensive as the available platform types are always growing.

See also:

Writing Custom Platform Types which explains the Platform API in detail.

View page

− Doors

Doors are a special platform type that can be opened and closed by a key item (see also Items).

Doors have the following properties:

Doors have the following properties:

Property	Description
keyType	Identifier matching the 'type' value of a Key Item. If empty the door does not require a key, if not empty the user must possess the matching key to open the door.
startOpen	Boolean indicating if the door starts open or closed (true for open).

Enterable Door

The enterable door is the most common type of door. It allows you to configure additional controls for entering the door:

Property	Description
doorEntryMethod	How do we enter the door, see below.
actionKey	If the entry method requires an action key, this sets the action key that should be pressed to enter.

The available entry methods are:

AUTOMATIC	The door is entered when the character stands on the door platform.
PRESS_UP	The door is entered when the character presses up.
PRESS_ACTION_KEY	The door is entered when the user presses the key defined by the actionKey setting.

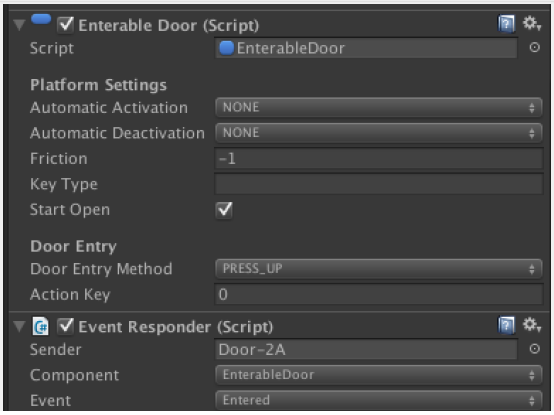
***Note**: you can only enter an OPEN door.*

Doors Send Events

Most doors types do not do anything to the character other than sending events. If you wish to do something when the door changes state (for example when it opens or when the character enters it) then you should respond to the door events.

Door Event	Sent By	Sent When
Open	All Doors	When the door is opened.
Closed	All Doors	When the door is closed.
Enter	Enterable Doors	When the door is entered.

The most common pattern is to create an *EnterableDoor* and respond to the *Enter* event by respawning the character in a new position or in a new level. This setup looks like this:



***Note**: In order to respawn at a new position you must also create a RespawnPoint (see Respawning).*

You can use the templates: *Template_EnterableDoor* or *Template_EnterableDoorNewLevel* to quickly add doors.

View page

– Writing Custom Platform Types

With a little coding knowledge you can easily create your own platform types by extending the existing Platform types. The API has several key elements to consider. Full API doc is available online here.

***Tip**: If you don't have coding skills you can always contact JNA Mobile and ask us to build your custom platforms. Although we can't guarantee a solution we are always eager to help and extend the capabilities of Platformer PRO.*

Collisions

The basic task of a custom platform is to respond to collisions. The collision routine is broken in to three functions:

virtual public bool Collide(PlatformCollisionArgs args)

The entry point for the characters interaction with the platform. Returns true if the character should be parented to the platform. By default it does the following:

```
virtual public bool Collide(PlatformCollisionArgs args) {
  BaseCollide(args);
  return CustomCollide(args);
}
```

For most Platforms you will not need to change this.

protected void BaseCollide(PlatformCollisionArgs args)

The built-in handling of collisions. Cannot be overridden but if you don't want default behaviours you can override the *Collide*() function so that this is not called.

virtual protected bool CustomCollide(PlatformCollisionArgs args)

Most of your custom platform code should go here. By default the value that this function returns will also be returned by *Collide*().

Typically your custom collision will check the type of collider that is hitting the platform and if the condition is met it will take some action. For example a moving platform will generally check if the characters FOOT is hitting the platform and if so will parent the character:

```
override protected bool CustomCollide(PlatformCollisionArgs args)
{
  if (args.RaycastCollider.RaycastType == RaycastType.FOOT)
  {
    return true;
  }
  return false;
}
```

The PlatformCollisionArgs (see API doc) class includes a reference to the raycast details of the penetration of the collider. You can for example

The *PlatformCollisionArgs* (see API doc) also include a Character reference and details of the penetration of the collider. You can for example use these to interact with the character. For example to freeze the character in the air when they move through a platform:

```
override protected bool CustomCollide(PlatformCollisionArgs args)
{
  args.Character.SetVelocityY(0);
  return false;
}
```

Skip Movement

The skip movement function:

virtual public bool SkipMovement(Character character, Movement movement)

is called by the Character each frame and allows a Platform to disallow certain movement types. It is passed the movement type being evaluation by *Character.TransitionMovement* and if the function returns true that movement will not be used.

For example a sticky substance which stops you jumping could do the following:

```
override public bool SkipMovement(Character character, Movement movement)
{
  if (movement is AirMovement) {
    return true;
  }
  return false;
}
```

***Note**: The default air and ground movements cannot be skipped.*

Parent and Unparent

When a character is parented to the platform the *Parent*() function is called. You can use this to respond to the parenting. For example you may start a timer which will cause the character to die if they don't unparent in time:

```
override public void Parent()
{
  StartCoroutine(DieInFiveSeconds());
}
```

Similarly when a character is unparented from a Platform the *UnParent*() function is called. You can use this to respond to the unparenting. For example you may stop the timer from the above example:

```
override public void UnParent()
{
  StopCoroutine(DieInFiveSeconds);
}
```

View page

Triggers and Event Responders

– Triggers

Platformer PRO has its own trigger system which can be used to trigger changes in your game

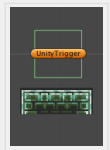
Platformer PRO has its own trigger system which can be used to trigger changes in your game.

You can also get started with triggers by following the intro tutorial:

https://www.youtube.com/watch?v=5wj3dfp-5Gs

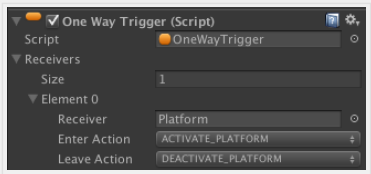
Trigger Types

The following trigger types are available:

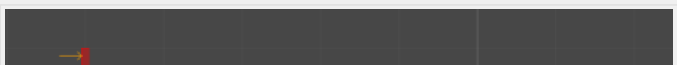
Type	Description
Proximity Trigger 	Triggers effects when the Character gets within a certain distance (shown by the Orange circle).
One-way Trigger 	Triggers effect when the Character walks through it from the direction indicated by the arrows.
Trigger Platform 	A platform that also acts as a trigger. Can be set up to respond to a character landing on it or when a character headbutts it.
Unity Trigger 	A wrapper around a standard Unity Collider2D set up as a trigger. Triggers the same way as a Unity trigger but also allows you to easily attach platforms and have conditions like requiring an item. For consistency we recommend you use the Unity Trigger wrapper instead of directly using a Unity Trigger but this is not required.

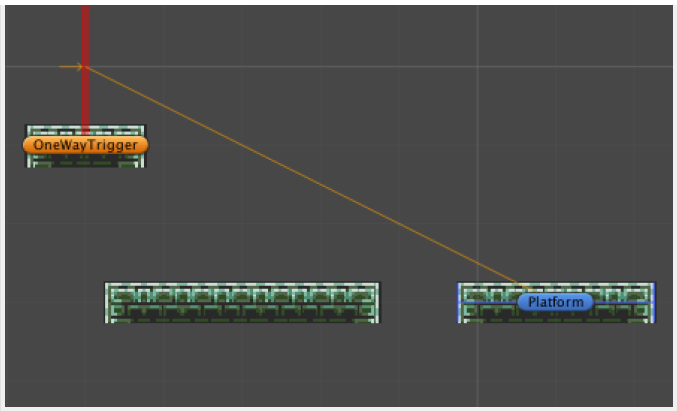
Trigger Receivers

Triggers can automatically do something to one or more receivers. Receivers are configured by adding a list item and dropping a GameObject or Component reference in to the *Receiver* field.



Once a receiver is added the scene view will show a line from the trigger to the receiver (assuming gizmos are on for the trigger):





Trigger Actions

Once connected to a receiver the trigger can perform various actions on the receiver when the trigger is entered or left. The following actions are available:

Action	Usage	Receiver Type*
SEND_MESSAGE	Sends a message to a GameObject.	GameObject
ACTIVATE_PLATFORM	Turns a platform on.	Platform
DEACTIVATE_PLATFORM	Turns a platform off.	Platform
ACTIVATE_GAMEOBJECT	Activates a GameObject.	GameObject
DEACTIVATE_GAMEOBJECT	Deactivates a GameObject.	GameObject
CHANGE_CAMERAZONE	Triggers a camera zone switch (with associated animation if configured).	PlatformCamera
SWITCH_SPRITE	Changes a sprite.	SpriteRenderer
SHOW_DIALOG	Shows a dialog.	Dialog
HIDE_DIALOG	Hides a dialog.	Dialog
OPEN_DOOR	Opens a door.	Door
CLOSE_DOOR	Closes a door.	Door
ACTIVATE_SPAWN_POINT	Activates a spawn point.	R
PLAY_ANIMATION	Plays an animation.	Animator.

*Note that the reciever is always a GameObject. When a component type is listed it means the GameObject must have a component of the given type.

***Note**: For more options see Trigger Events below.*

Trigger Conditions

Most triggers allow you to apply a conditions so that they only activate when the conditions are met.

Required Item - Character must have an Inventory and the Inventory must contain an item which has a type that is an exact match of *Required Item* or the trigger will not trigger.

Required Component - The referenced component must be present, active and enabled, or the trigger will not trigger. You can use this, for example, to have triggers that only respond to characters with specific movements enabled, for example only trigger when the Character has their weapon out.

***Note**: Some triggers have additional conditions, for example a one-way trigger allows you to specify a direction and speed via the Velocity field.*

Other Trigger Options

One Shot - Trigger will only fire once.

Auto Leave time - After the trigger is entered, wait *Auto Leave Time* amount of seconds and then automatically trigger the leave actions, even if the character has not left the trigger. A non-zero value will also prevent the leave action being triggered when the character leaves the trigger.

Trigger Events

If you wish to do something else (i.e. somethinbg that is not available in the tirgger actions) when a trigger is entered or left you can use an *Event Responder* in conjunction with the trigger. Triggers send the following events:

TriggerEntered - Fired when trigger is entered.

TriggerExited - Fired when trigger is left.

Unity Triggers

Nothing prevents you from using standard Unity triggers in your game. We suggest that you add the UnityTrigger wrapper to the GameObject the has your Collider2D on it: this enables you to use the rest of the trigger capabilities while still using Unity triggers. You do not have to do this.

Typically you do not need to add a special collider to the Character to trigger the Unity triggers, because most characters already have a HurtBox collider configured. Keep in mind that the layer of your trigger must be configured in the the Physics2D settings to collide with the layer of your characters collider.

The recommended layers are *Character* for the character hurt box and *Collectable_Projectile* for the trigger.

***Remember**: By default raycasts hit triggers (its a Unity Physics2D setting) and that means that your character will not be able to walk through triggers that are in the geometry layer!*

View page

– Event Responders

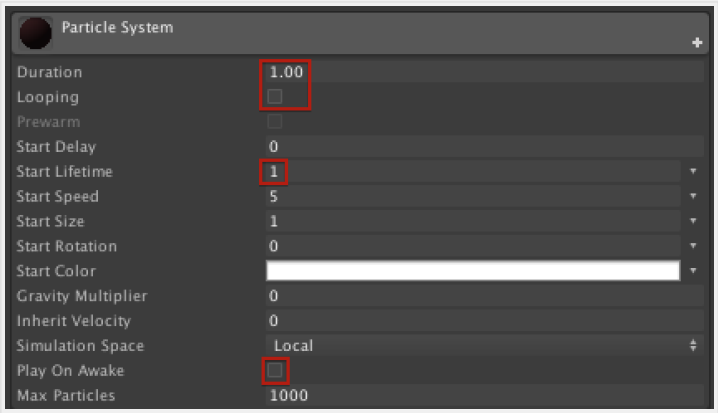
Event responders enable you to respond to c# events thrown by Components, and are typically used for things like particle effects and animations. Almost all components in Platformer Pro raise events so you can easily attach effects (or other behaviour) to these components.

***Tip**: You can also use EventResponders to listen and respond to events thrown by third-party scripts.*

Quick Start - Responding to Damage

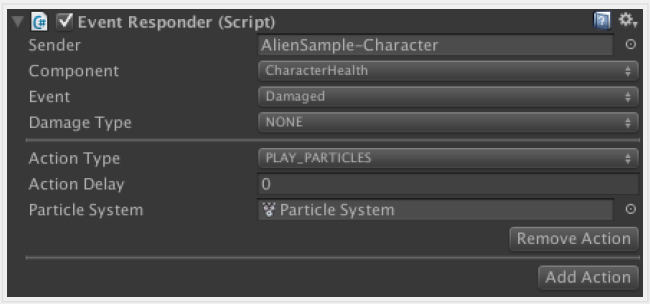
These steps assumes you have a character with a CharacterHealth component already set up in a scene. Make sure your character has more than 1 health.

1. Create a new empty *GameObject* as a child of your Character.
2. Rename this new *GameObject* to 'DamageEvent'.
3. Create a particle system and drag it so it as a child of the *DamageEvent GameObject*
4. Set up your *Particle* system: set *Loop* and *Play on Awake* to false. Change the duration and *Start Lifetime* to 1.
(Make sure you can see your particle system ... the default particle system will render behind sprites, so you may want to change the material or shader).



5. Add an *EventResponder* component to the *DamageEvent GameObject*.

6. Drag your *Character GameObject* in to the *Sender* field of the *EventResponder*.
7. Select *CharacterHealth* as the *EventResponder*
8. Set *Damaged* as the event and leave the *DamageType* set to *NONE*.
9. Click the *Add Action* button.
10. Choose *PLAY_PARTICLES* form the *Action* drop down list.
11. Drop your particle system from above in to the *Particle System* field.



Press play and cause your player some damage. The particle system will be played.

See *Available Events* for a list of available events.

Event Filters

Some events allow a filter to be assigned which can be used to only respond to the event if the conditions are met.

Damage Type - If you use damage type on your damage then you can set up different events based on the damage type. For example you could play fire particles when the damage type is FIRE, or ice particles when the damage type is COLD.

AnimationState - Filter the events by animation state to play particles based on the characters current animation.

ButtonEventArgs - Sent by touch buttons you can filter the response based on the state (DOWN, HELD, UP).

Action Types

There are many action types available:

DEBUG_LOG - Logs a message to the console. Useful for testing.
SEND_MESSSAGE - Sends a user defined message to a Component. Use to invoke your own scripts.
ACTIVATE_GAMEOBJECT - Activates the given game object.
DEACTIVATE_GAMEOBJECT - Deactivates the given game object.
ENABLE_BEHAVIOUR - Enables the given behaviour.
DISABLE_BEHAVIOUR - Disables the given behaviour.
OVERRIDE_ANIMATON - Sets an animation override on the selected character.
CLEAR_ANIMATION_OVERRIDE - Clears all overrides on the selected character.
PLAY_PARTICLES - Plays a ParticleSystem.
PAUSE_PARTICLES - Pauses a ParticleSystem.
SWITCH_SPRITE - Switches the sprite on a sprite renderer.
PLAY_SFX - Plays a sound effect (only for audio set up using SFX system).
PLAY_SONG - Plays a song (only for audio set up using SONG system).
STOP_SONG - Stops a song (only for audio set up using SONG system).
MAKE_INVULNERABLE - Makes character invulnerable for period of time.
MAKE_VULNERABLE - Makes character vulnerable again.
LEVEL_COMPLETE - Completes the level triggering level complete events.
LOAD_SCENE - Load the scene with the given name (scene must be in Build Settings).
LOCK - Lock the provided level.
UNLOCK - Unlock the provided level.
RESPAWN - Respawn the character at the given respawn point.
SET_ACTIVE_RESPAWN - Set the active respawn point.
START_EFFECT - Starts playing an effect set up with the FX system.
FLIP_GRAVITY - Flips gravity (requires character to supportFlippableGravity).
PLAY_ANIMATION - Plays an animation using a SpecialMovement_PlayAnimation component.
START_SWIM - Starts the character swimming, use when entering water.
STOP_SWIM - Stops the character swimming, use when exiting water.
ADD_SCORE - Adds the given value to the given score type.
RESET_SCORE - Resets the given score type to zero.

Action Delay

You can delay an action so that it doesn't take affect immediately. For example you may want to delay the character Respawn to give time for a particle

effect to finish.

Warning: If you use many action delays you will notice a small garbage collection overhead. This is typically okay, but on mobile devices you may want to avoid action delays on events that occur regularly (like animation state changes).

Available Events

The EventResponder can listen to any C# event that has event arguments that derive from System.EventArgs.

The following events are send by Platformer Pro components:

CharacterHealth
Loaded - Fired when the health information is loaded from the saved data.
Healed - Fired when the Characters health is increased.
Damaged - Fired when the Characters health is reduced.
Died - Fired when the Character dies.
GameOver - Fired when the Game ends.

Character
ChangeAnimationState - Fired when animation state changed. Used by animation bridges to animate character, but can also be used to drive particle effects or sound effects (see also Sound Effects Bridge).
Respawned - Fired when the Character has respawned at a new location.
WillExitScene - Fired when the Character is about to leave the scene. Often used for clean-up.

CharacterLoader
CharacterLoaded - Fired when the Character has been loaded.

TimeManager
GamePaused - Fired when the game is paused.
GameUnPaused - Fired when the game unpaused.

Enemy
ChangeAnimationState - Fired when animation state changed. Used by animation bridges to animate enemy.
CharacterDamaged - Fired when the Enemy causes damage to the player.
Collided - Fired when the Enemy runs in to a wall.
Damaged - Fired when the Enemy is damaged.
Died - Fired when the Enemy dies.

Item
CollectItem - Triggered when the item is collected.

Door
Open - Fired when the door is opened (triggers when open starts).
Closed - Fired when the door is closed (triggers when close starts).
Entered - Fired when the character enters the door (EnterableDoor only).

Triggers
TriggerEntered - Fired when a trigger is entered.
TriggerExited - Fired when a trigger is exited.

LevelManager
LevelComplete - Character finished the level.
Loaded - Level manager data finished loading.
Respawned - Character was respawned.

SceneLoaded - Scene finished loading.

***Note**: this list is not exhaustive as we add new events all the time.*

View page

UI and Effects

– UI Overview

Platformer PRO includes a number of UI scripts and components which make it possible to get a user interface in to your game with zero coding.

Even if you wish to use your own UI components, you may want to browse the code of the included UI components to get ideas about the best way to connect UI to Platformer PRO.

There are several classes of UI components:

Menu System - A system for defining menus.

UI Screens - Full screen overlays shown on start, pause, or game over.

UI In-game Components - Small components that show in game details like the number of lives a player has or the items a player has collected.

Level Select Screen - Found in the Alien sample this allows you to build a SMB3 like level select screen.

Platformer PRO also includes a simple **tweening** and **FX system** which can help add some visual flair to both the UI and to in-game components.

View page

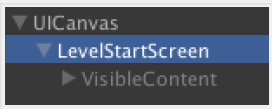
– UI Level Start Screen

The level start screen is a full screen overview that can be shown when a scene loads or when the character respawns.

Basic steps for creating a start screen:

1. Create a UI canvas (or use an existing one).
2. Create an Empty GameObject as a child of the canvas.
3. Add the *UILevel_Start_Screen* component to the Empty GameObject.
4. Rename this GameObject 'UIStartScreen' (not required but useful).
5. Create an Empty GameObject as a child of the UIStartScreen GameObject.
6. Rename this GameObject 'VisibleContent' (not required but useful).
7. Assign the VisibleContent GamObject to the Visible Component property of the UILevel_start_Screen component.
8. Deactivate the VisibleContent GameObject.

The final product should look something like this:



Properties

UILevel_Start_Screen has the following properties:

Visible Content	GameObject that will be activated, deactivated to show/hide the menu.
Show on Start	If true this will be shown on start.
Show on Respawn	If true this be shown on character respawn.
Total Show Time	How long should the screen be shown for.
	A list of effects that will be used to show the start screen (for example

Show Effects	fade in).
Hide Effects	A list of effects that will be used to hide the start screen (for example fade out). Note that if you have no hide effects the VisibleContent GameObject will be deactivated automatically, BUT if you have hide effects the VisibleContent GameObject will not be deactivated: the hide effects must do the job of hiding this component.

Adding Content

You can add any visible content such as images or text as children of the *VisibleContent* GameObject.

***Note**: This screen is quite a simple screen, you may wish to extend or replace with your own implementation.*

View page

Extending and Using Third Party Tools

Platformer PRO - Troubleshooting

Compilation Issues and Bugs

– Jetpack as Double Jump not working in Unity 5

Affects: Version 1.0.2 and below.

Issue

The Jetpack as Double Jump is in the wrong namespace which raises an error in the console and does not work.

Resolution

To fix find this code (at the start of *AirMovement_DoubleJumpJetpack.cs*):

```
namespace PlatformerPro.Experimental
```

and change it to:

```
namespace PlatformerPro
```

View page

– CommandBro and Pixtroid Samples crashing in Unity 5.2

There seems to be a bug in Unity 5.2 when using AnimatorOverrideControllers. These are used in CommandBro and Pixtroid samples for shoot animations.

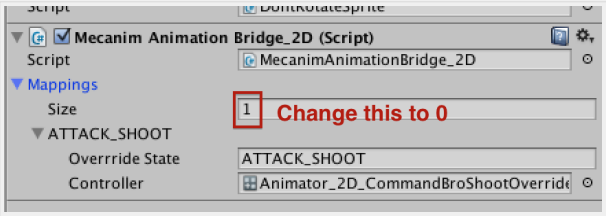
We are currently investigating but at this stage suggest you:

1. Use Unity 5.1 (this version seems to be a lot more stable in general).

2. If you must use 5.2 don't use animation overrides (for example instead set the AnimationState in the Attack movement).

This page will be updated when a fix is available (if possible, as mentioned it seems to be a Unity bug).

If you want to play the samples in 5.2 you can disable the overrides which will mean everything will work, except the shoot animations:



View page

– I get a compilation error when I try to build

Occasionally in my zeal to get things out I don't build to all platforms. Here are some fixes that you can apply until the patch is out.

v1.0.1

Issue: The AirMovement_Float doesn't wrap the draw inspector call in #IF UNITY_EDITOR tags.

To fix, change:

```
        /// <summary>
        /// Draws the inspector.
        /// </summary>
        new public static MovementVariable[] DrawInspector(MovementVariable[] movementData, ref bool showDetails, Character target)
        {
            // Same as base class.
            return AirMovement_JetPack.DrawInspector (movementData, ref showDetails, target);
        }
```

To:

```
#if UNITY_EDITOR
        /// <summary>
        /// Draws the inspector.
        /// </summary>
        new public static MovementVariable[] DrawInspector(MovementVariable[] movementData, ref bool showDetails, Character target)
        {
            // Same as base class.
            return AirMovement_JetPack.DrawInspector (movementData, ref showDetails, target);
        }
#endif
```

View page

Main Characters

– Character can't collect Item with a KeyNotFoundException

If you see the following error message:

It means your saved ItemData is not inline with your Characters ItemManager. This can either be because the ItemManager has changed, or can happen when upgrading versions.

To fix simply reset the ItemData by resetting player prefs, from the menu select **Asset->Platformer PRO-> Persistence->Reset all Player Prefs**

View page

Enemies

Animation

Platforms

– My character bounces between climb and fall when I hold UP on a ladder

If you have a dismount of TOP/BOTTOM on your ladder then the character requires a Passthrough Platform to stand on.

If you take a look at the Ladder in the CommandBro sample you will see a sample ladder with a passthrough.

To fix this problem:

- Add a passthrough platform with an edge collider (preferred) or Box2D collider.

- Ensure the top edge of this passthrough platform collider aligns with or is slightly above the Ladders BoxCollider2D

- - -

If you still see the issue its likely your 'look ahead' settings aren't correct. Platformer PRO uses the feet look ahead to find ladders (and other platforms), and the grounded look ahead to find the ground.

Your feet look ahead should be larger than your grounded look ahead and typically it is MUCH larger. To fix:

- Go to Character Inspector and open the Advacned Settings.

- Set Feet Look Ahead to 0.3f*

- Set Grounded Look Ahead to 0.1f*

*Note this change may affect items like non-rotating slope behaviour, landing behaviour, etc. The values here are just a suggestion.

View page

RECENT ACTIVITY

No recent activities yet.