USER MANUAL

By Lightbug

Note: This document will introduce you to the Kinematic 2D (K2D for short) world. For future releases and updates this document will be slowly converted into a proper online “user manual” (along with an online API Reference) with every relevant detail of the package.

Also, I’m doing my best with the English I know 😊, so forgive me if there are a few mistakes here and there, eventually I will fix them.

Contact: lightbug14@gmail.com

Introduction / FAQ

What is Kinematic 2D?

Kinematic 2D is a Fully Kinematic (non-physics based) 2D Character controller solution that allows you to do the movement of your 2D character while handling collisions.

What is not Kinematic 2D?

Kinematic 2D is not a template/system where you create your character based on predefined components, specific abilities, states, audio systems, game score, etc. If your goal is to make a game as fast as possible, using drag and drop components to your characters this is probably not the asset for you.

If this package contains some of these features is because i chose to add them merely as examples, or extras/helpers.

By 2D do you mean sprites?

No, by 2D i mean X and Y coordinates. All the collision detection algorithms and movements involved are based on these coordinates. Also there is no need to work in an axis aligned environment, the character should work fine in any orientation.

What types of games this package supports?

This asset was initially designed for 2D platform games (platformers) in general, although you can use it in some clever way and make it work the way you want.

What is a Character controller?

A character controller is a component that gives you full control regarding your character movement, state, pose, animation, you call it. The term “controller” has many meanings nowadays, some may refer to a character controller exclusively to the movement aspect, some to the movement and animation, collision, etc.

In this case i will be referring to a “Character controller” as a component that make use of the functionality of the component responsible for collision detection and proper movement/rotation of the character.

What Kinematic means?

Unlike dynamic rigidbodies, kinematic rigidbodies are objects that are not affected by the Physics system, this means that their movement and collision detection has to be manually scripted frame by frame 😳… The great advantage of doing this is that it gives you full control over your character movement behaviour.

But why Kinematic? The default dynamic rigidbody already do collision detection?

Yes it does, but let me ask you, have you try to create a character based on a dynamic rigidbody and see that the behaviour becomes at some point unpredictable? maybe it's fine for small things but when you try to add slopes, stairs, or even craft unnatural movement behaviours, at some point you will end up tweaking a lot just for the character to behave exactly like you want. Here, this is an original tweet from **Team Cherry** 🙌 , creators of *Hollow Knight*, quote:

"*We use the inbuilt rigidbody2d component and gravity, yeah, although we do a lot of tweaking to make the player feel nice and snappy in classic megaman-style way*"

(https://twitter.com/teamcherrygames/status/946171459440803840)

If you want to control each and every single aspect of your character movement and rotation (and not die in the attempt) the kinematic solution is the answer.

Remember, a character is not a "real" object, do not relate a dynamic rigidbody with a real human (common mistake), you (as a person) do a lot of stuff just to be standing up, move, turn around, etc., a dynamic rigidbody do not.

Could you please consider building X feature for the Character Controller?

I don't promise anything, but if I consider the new feature is somehow "needed" or that will improve the overall feeling of the Character Controller sure, i'll give it a try.

What about a new feature for the Character Motor?

This part is more complicated, and probably not so "open to the public", but yes, if you propose some feature maybe i will consider it.

Man, I just have uploaded the package and suddenly my whole game is broken. What happened?

Everytime you download/import an update do it with caution (probably this is a general advice, not limited to this package). Try to peek at the new version of Kinematic 2D inside an empty project, take a look at the content, check if something is there, it is possible that i have missed some important content, it could happen.

This is my way of versioning the package:

Major.Feature.Minor

- Minor: Bug fixes, tiny changes, code improvements, etc. This update probably won't affect your current project, and it's always recommended.
- Feature: Features addition and updates, probably this comes with a new Character controller ability or state, a new way of doing things internally in the character motor, or a new fancy component. Beware! It could break some things.
- Major: This type of update will for sure break almost everything, this means a new fresh start, new systems, new components, an overall change and improvement of the code structure, new asset folder structure, etc.

Unity version

Supported versions

K2D(from version 2.1.1) is officially supported from version **5.6.7 or higher**. It is intended to keep uploading this package using two Unity versions: 5.6.7 and 2018.3.2.

Unofficial supported versions

The core functionality of this package is prepared to work from **5.3.0 onwards**, this is because 5.3.0 introduced the BoxCast methods for both 2D and 3D worlds. So, **Why am i uploading K2D with 5.6.7 instead of 5.3.0?**

5.6.0 ≤ version < 5.6.7
The only problem with these versions is that the editor does crash everytime a scene(from K2D) is open, at least this is my experience on windows 10. The safest version to upload the package is (by the moment i'm writing these words) 5.6.7.

5.3.0 ≤ version < 5.6.0
Well, trying to carry on the whole package with all the scenes from 2018.3.2 to Unity 5.3.0 was a mess, the assets (sprites, prefabs, textures, models, etc) were completely butchered, the scenes couldn't open (same as 5.6.0), bugs all over the place presumably due to Unity backwards compatibility issues. The bright side of this is that the functionality of the package works well (all the components and the code).

This doesn't mean that you couldn't use K2D as a tool in this unsupported versions (5.6.0 or even 5.3.0) the core functionality work. In any case i would recommend to directly ignore the Walkthrough (or better yet delete the entire folder), also delete the prefabs (the three characters)

Differences between Kinematic 2D versions

If your version is 2018.3.2 or higher you will be getting all the nice features (editor-wise) that Unity 2018.3 brings, some really good features, here are two of the most important (IMO):

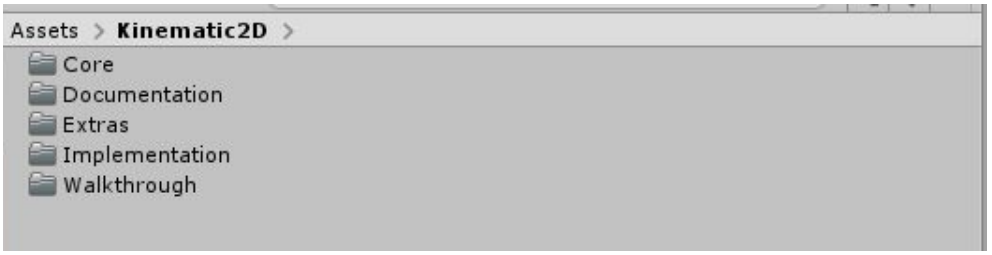
- Prefab system: Although i loved the new prefab system, backward compatibility (to version 2017.4) was a pain in the a$$, totally broken. Not to mention that the new override system is great, allowing me to create prefabs more quickly, specially on an asset like this one, where the component inspector is full of settings, layermask and fields.

- Project Presets: Great addition to the editor 👍, the problem regarding the layers names (see next section) goes away with two clicks.

Of course the whole package works the same regardless of the version used, but for users with Unity 2018.3 or higher the initial settings configuration is a piece of cake (Inputs and Layers). The rest should behave exactly the same.

Importing the package

Once you have imported Kinematic 2D to your project it will appear a folder in the root directory(“Assets”) called “Kinematic2D”, inside will look like this:



It is highly recommended to go first to “Kinematic 2D/Documentation/ReadMe” and then to the “Walkthrough” folder, in there you will find all the scenes related to this package.

Note: On previous versions (<2.0.0) there was available a “Kinematic 2D/Demo/Scenes” folder, those scenes were removed from the package, instead all the needed scenes to learn this assets are located at “Kinematic 2D/Walkthrough/Scenes”.

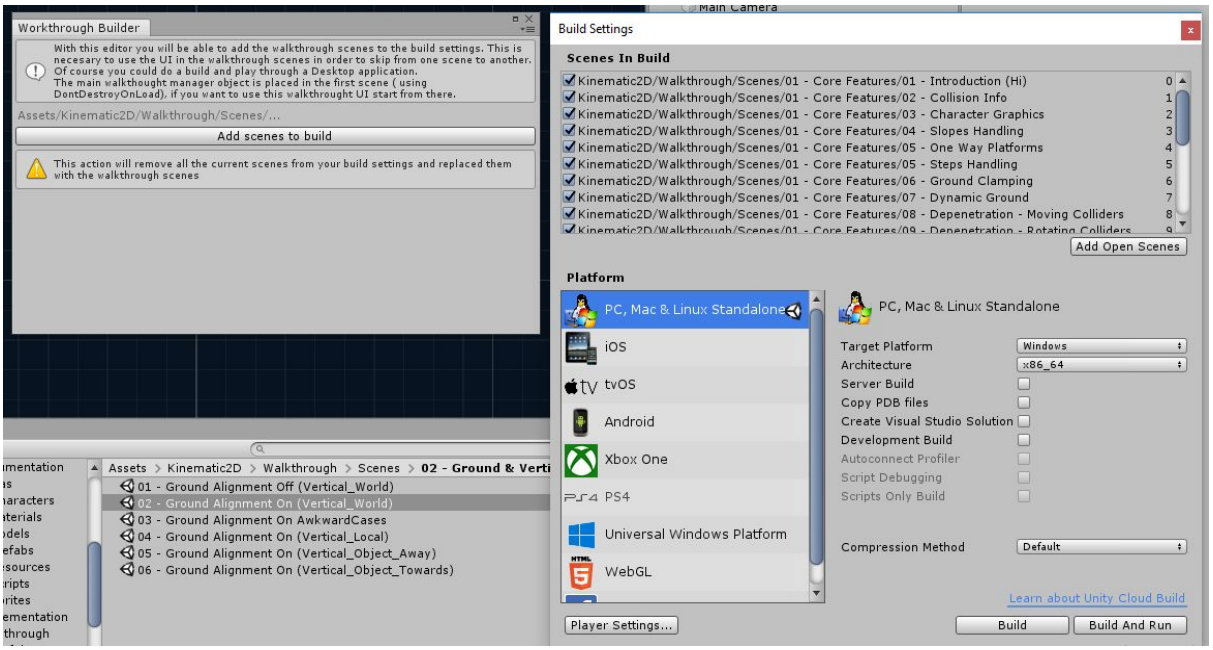
Walkthrough

In K2D is included a typical “Walkthrough section” with a bunch of small and concise scenes, created with the goal of introduce the package. I recommend you to go through all of the scenes in order to learn and test some of the features.

Because having to manually pass from one scene to another is painful (unless you want to play with the gameObjects of the scene) i created a simple “Walkthrough Builder”, this editor script automatically search for a given path, takes all the scenes in there (recursively) and add them to the build settings scenes list.

So, **Why this “Builder” exist in the first place?** In order to move from one scene to another (through buttons) is required by Unity to have all the scenes involved added to the build settings scenes list.

The Walkthrough Builder editor can be access at “*Window/Kinematic 2D/Walkthrough Builder*”.



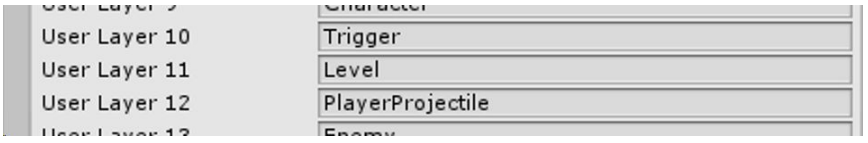
Once this is done you could use the UI from the first scene on the unity editor or just make the build, your choice.

The layers problem

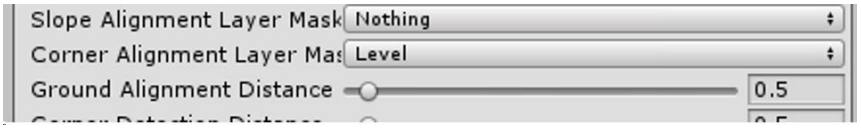
Here is the thing, you will **probably** see all the layer masks available as empty slots, in reality there are not empty, the layers system in Unity works with integers, limited by the number of layers the engine can handle (32 layers).

Why am I telling you about this if everything is going to work just fine? Because it could happen that you already have names for those layers (and the Layer masks fields will surely look very strange).

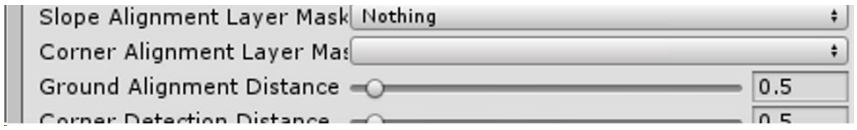
For example: in my case the layer 11 is called “Level” (at the moment I’m writing this document), it looks like this:



So, a character’s inspector that has this “Level” layer inside any layer mask looks like this:



Let’s say that layer 11 is empty (not name assigned) inside your project settings, so, the same scene as before will look like this:



If you open the layer mask list **all unnamed layers will not appear.**

So, What should you do?
First, don't worry about this behavior, it should not affect the demo scene nor your project. The obvious downside is that you won't get to see the layers Mask settings, and that's bad.

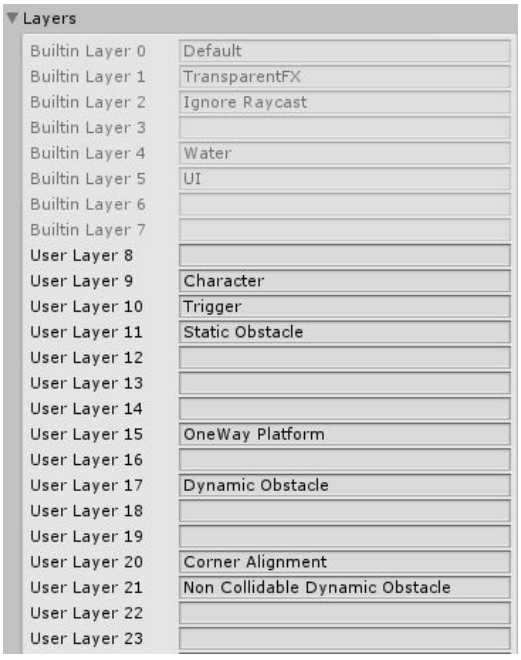
The clear solution is to replicate the exact same layers names that i have, here is how:

Solution 1: Using predefined "Presets" (**≥ 2018.1**)

If your current Unity Editor supports "presets" (mentioned in the Unity Version section from above) go to the layers inspector and load the "Kinematic2D_Layers" asset (placed at "*Kinematic2D/Documentation/Project Presets*"). If you want to preserve your original layers name remember to save you own layers in a preset (in case you want to go back).

Solution 2: Manually naming the layers (< 2018.1)

Here are all the Layers from the project:

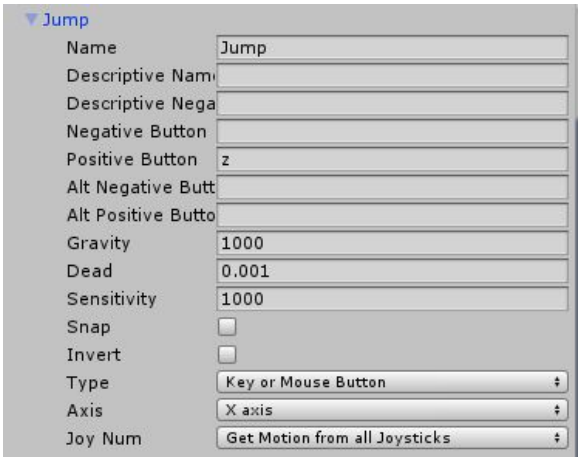


Default Input settings (from last released version)

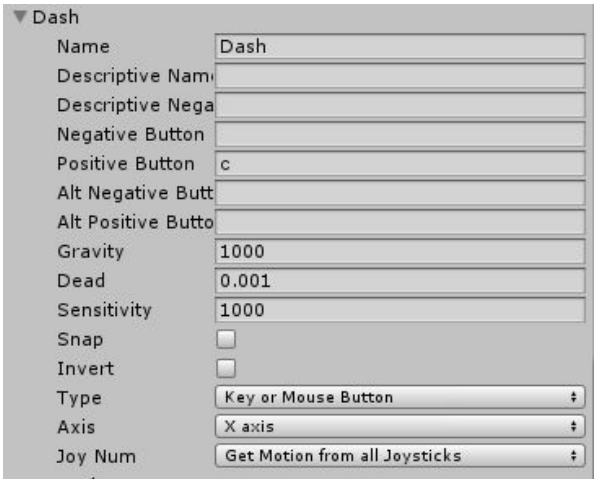
K2D uses a slightly modified default input settings. The only additions were (Except “Jump”):

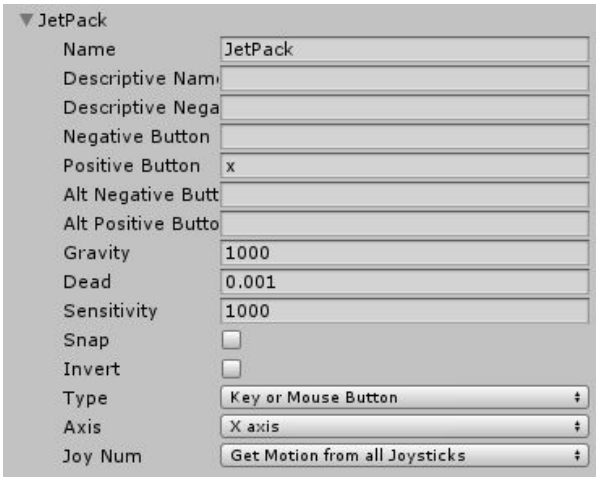
Again, if your editor supports Presets use instead the presets provided with the package. If you have to manually setup the inputs here is how:

1 - Modify the default Jump input, change only the Positive Button from “space” to “z” (by default):

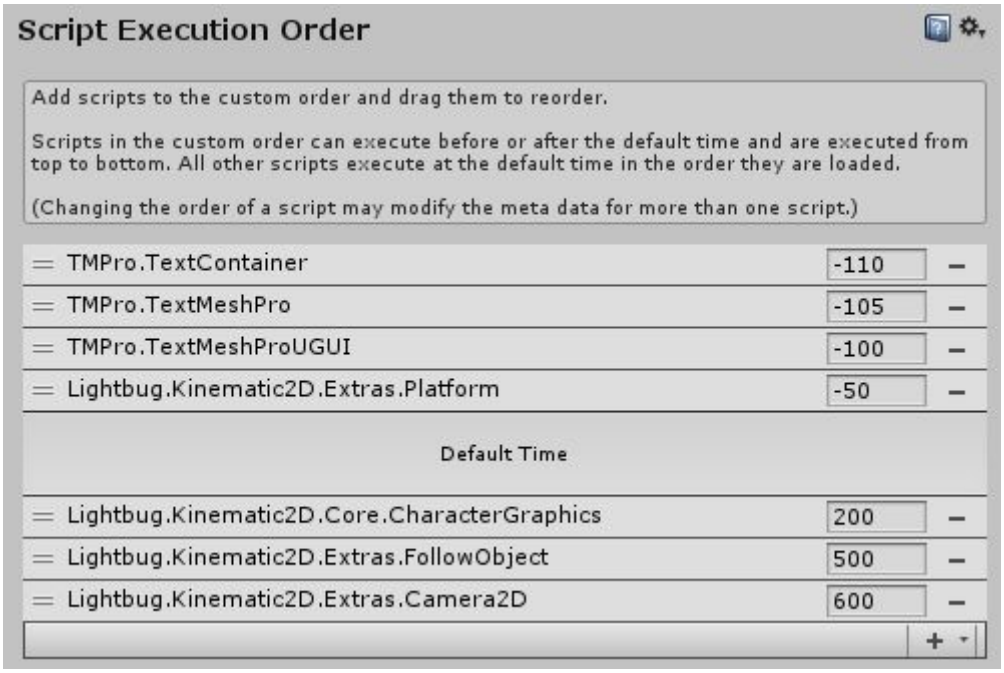


2 - Select the Jump input, right click -> Duplicate Array Element (x2) and repeat the process, this time with “Dash” (positive Button = c) and “JetPack” (positive Button = x):



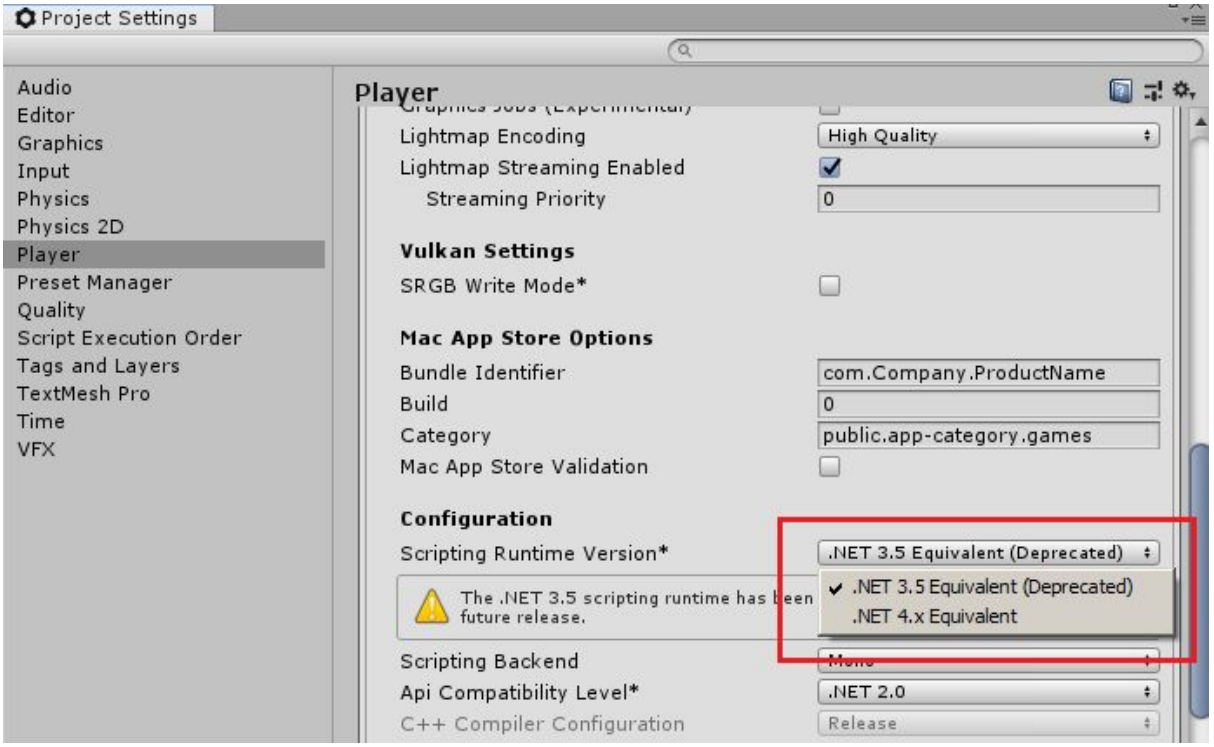


Default Script Execution Order(from last released version)



Missing script references

It could happen that all the scripts references might be lost when importing the package. If this happen don't worry, this is probably due to a scripting runtime version problem, switch between the two versions available (.NET 3.5 and .NET 4.x) and then restart the editor. In order to do this go to "Project Settings/Player".



Content

Core

There is a clear line between what i called the “core” and the “implementation”. Basically the core contains all the functionality related to the collision detection method and the movement/rotation procedure, basically it does the heavy lifting of the package, the “boring part”, nonetheless this is the most important part.

Implementation

The implementation consist of a bunch of components/scripts that implement the functionality of “the core”. These components may be related to the input management, scriptable objects, animation, movement, AI, etc.

This part of the package exists for the following reasons:

- You could learn from this scripts, for example how a simple character controller is made, how the character animation works, the AI system, the inputs system, etc.
- To provide you with a useful character controller for your future games, there are a lot of games that don’t even use the 20% of this components.
- To deliver a more complete package.
- To extend in future releases these functionalities.

Extras

Basically there are all the scripts and component that didn’t fit in the previous sections. This part is there just because the demo scenes required those components, for example, if you see a *“Camera2D”* component is there because at the moment i was making the scenes i needed a camera component. The same happened with the Platform component, the sine animation component, the rotation-shift component, the prefab instantiator component, demo managers and so on.

Creating a basic character

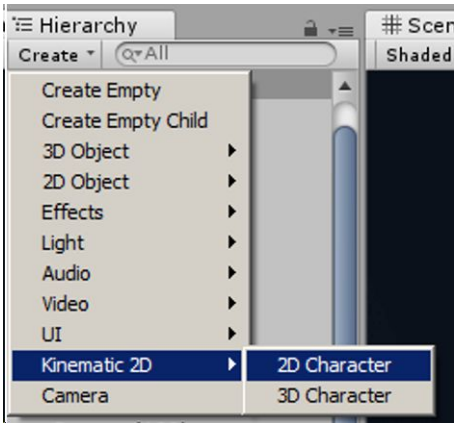
This instructions will help you to create a basic character that will not do anything, i know it does not sound very encouraging, but this is the first step and an important one.

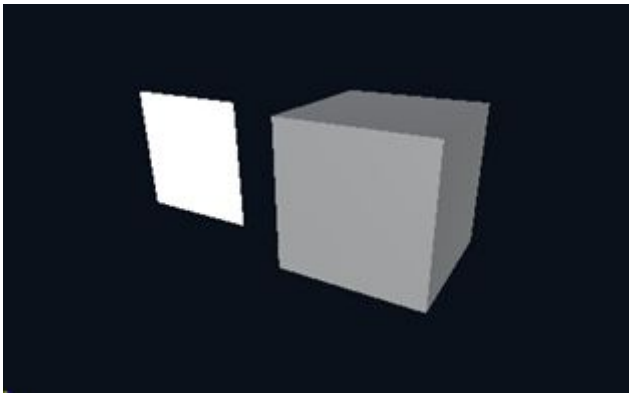
Steps:

1. Create an empty GameObject (The “Character”).
2. Add a *CharacterBody*2D (2D Physics) or CharacterBody3D (3D Physics) to the character.
   IMPORTANT: There is not need to add neither a *BoxCollider* nor a *Rigidbody* since the *Character Body* component is going to do this for you (setting kinematic options and interpolation stuff).
3. Add the CharacterMotor. (Check specially the LayerMask settings)
4. Add the visuals of your character (for example a sprite or 3d mesh) **as a child of the character**. This object acts as visual representation of your character.
5. Add to the visual object the “CharacterGraphics” component, this will allows you to separate the body shape (the root game Object) from the visuals of your character (the child Object). *[This becomes useful when the character should align itself to the ground or turn around.]*
6. Modify the Width and Height of the Character to fit your character visual GameObject.

Note: If you are planning to create a custom character controller i would recommend to derive your custom class from the CharacterMotor component.

[OPTIONAL] You can create an empty character (with a white square sprite for 2D, or a Box primitive for 3D) by selecting “Create/Kinematic 2D/Empty Character”.





Character body properties

For a detail explanation of any parameter see the attached "Tooltip" on the inspector. Any tooltip will appear by hovering the mouse cursor over the name of the field.

Body Transform

The character body possess a custom Transform, this is due to the multiple changes in position and rotation that the character may or may not suffer along the way. If you want to set some of the properties you can access the *BodyTransfrom* field from the *CharacterBody* component.

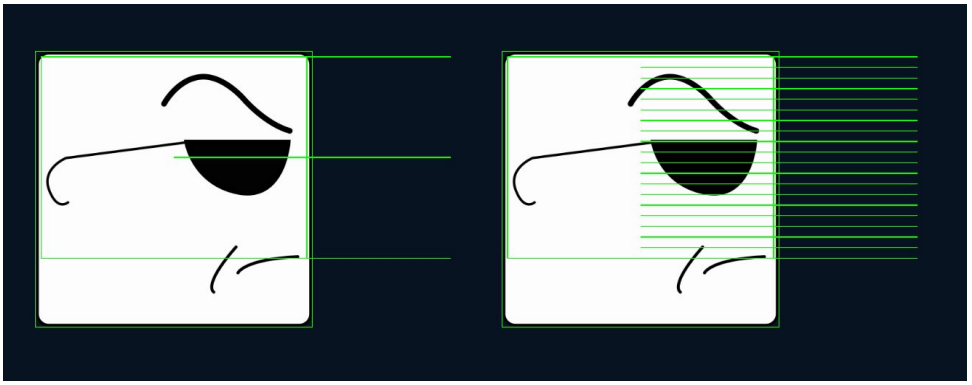
Note: Do not bother to modify the Unity transform directly, the character will pay attention only to the *BodyTransfrom* information.

Collision Detection Method

This asset uses the Raycast and the Boxcast methods (the best of both worlds) to solve all types of collisions, slope handing, depenetration, ground alignment, etc.

The *Boxcast* method make a sweep of an entire box shape from position A to position B, that is, it can sample any point that hits the box along all the area of the box.

The *Raycast* method behaves like a sampling laser array fired horizontally (just like the gizmos represent).



Skin width

The Skin Width is used to prevent the character from getting stuck with another collider. If you are experiencing some problems try to increase this value.

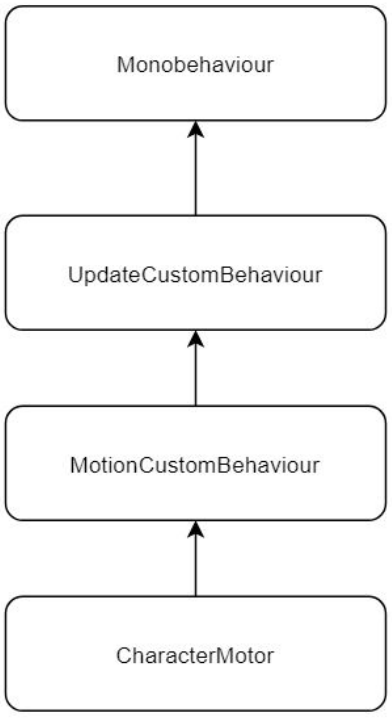
Step offset

The Step Offset determine the maximum step height the character can walk. This will also serves as a bottom skin width, so be aware of the value you assigned to it.

Character Motor

This section will explain the default behaviour of the Character Motor, its way of moving/rotating in the world, plus a brief overview of the update process behind.

Class structure



Monobehaviour: The old monobehaviour.

UpdateCustomBehaviour: This is basically a Monobehaviour wrapper, the only new thing this introduced is an enum field that allows you to select the update method of the Monobehaviour. The main reason to include this is because the update method can be change in the inspector. The clear downside is that all the unity messages are going to be called, although that's not really a performance issue.

MotionCustomBehaviour: Introduce all the functionally that's gonna handle the movement of the character body, either by using Transform.Translate/Rotate, Rigidbody.position/rotation or MovePosition/Rotation.

Movement methods

Velocity changing

The character motor will move internally frame by frame its own body based on the current **velocity** vector. This can be achieved by setting externally a velocity value, probably the most used way to move the character.

Move method

The move method sets the deltaPosition vector of the CharacterMotor component and do the movement handling collisions. Internally the update method of the Motor uses this method to do the movement, additionally you can call the Move method externally if you need to take into account collisions.

Teleport method

This involves an abrupt change in the *BodyTransform* information. This method is quite similar to Transform.Translate except it includes rotation as well.

Update process

De-penetration

The first step is to de-penetrate the character from overlapping colliders.

Dynamic ground movement

Checks the current ground information (stored internally in the character motor), if the character was grounded (and support dynamic ground movement) it performs the action required to move the character accordingly along with the ground, this involves a change in position and rotation as well.

Grounded/Not-Grounded Movement

If the character is in the *Grounded* or the *Not-grounded* state the action performed by the character motor will be the “Ground Movement” or the “Not Grounded Movement” respectively.
(See the next section for a more detail explanation)

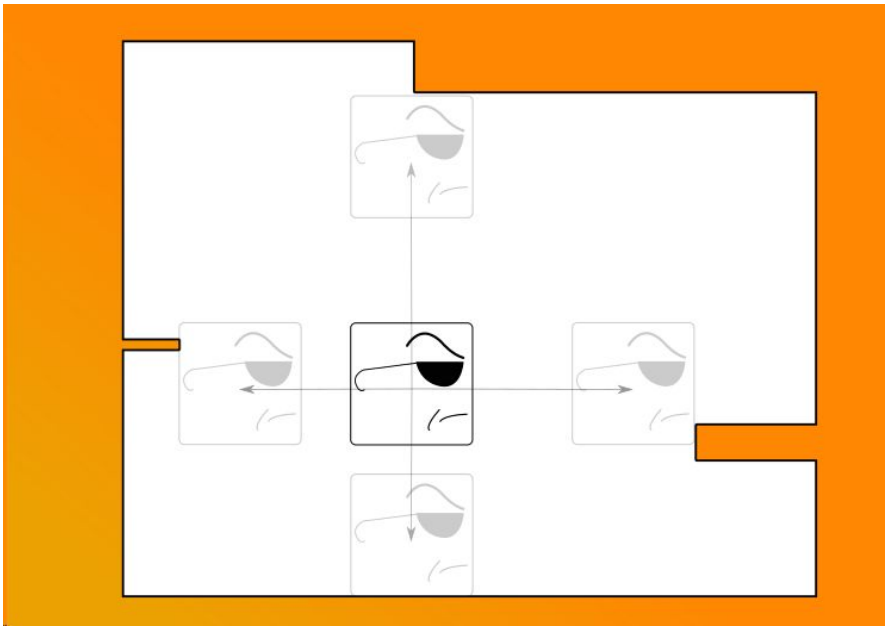
Move/Rotate the character

Once every collision and overlap has been solved the character will call the Movement and Rotation methods (based on the Motion Mode selected). This normally will happen once per frame.

Not-grounded movement

Movement Direction

The character motor translates the x component of the input vector directly to horizontal movement (towards the local right vector), same with the y component, but for vertical movement instead (towards the local up vector). Then projects the character body from point A to point B (desired position).
By projecting i mean that the component will do a collision detection method. Based on the information gathered the Character motor will move the body to the resolved location.



The only way out of this “not grounded” state is, like the name implies, when the character hits the ground, and enters the grounded state.

Effective body shape

The box shape projected by the collision detection algorithm corresponds to the whole body:

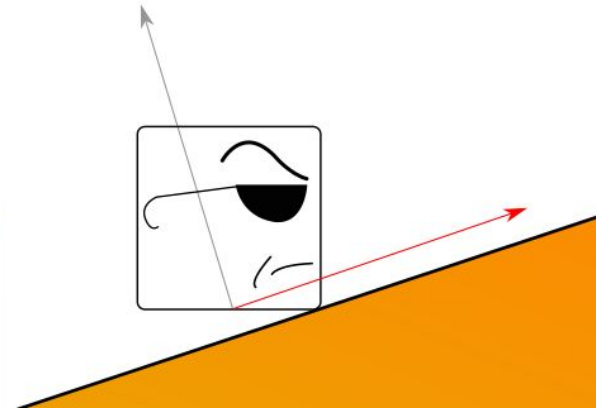
Grounded movement

The Grounded movement involves more steps that the not-grounded one.

Ground movement direction

Based on the input vector, the character motor translates the x component directly to a new direction called “ground movement direction”.

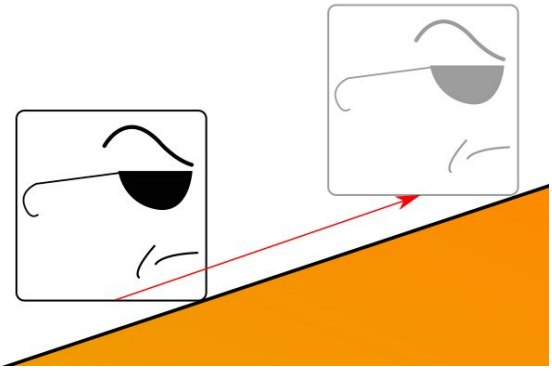
Here you can see this vector in color red.



Look that a positive input x component doesn’t match with the local right vector. So, the character could be climbing up a slope of 45 degrees while maintaining the input vector y component at zero.

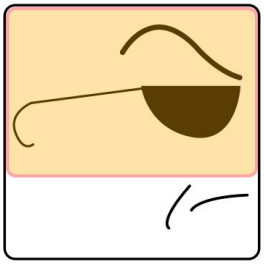
The grounded movement will always maintain the horizontal speed, it doesn’t matter the slope the character is climbing. This is due to the fact that the movement is applied towards the Ground movement direction vector.

Input = < speed * dt , 0 >



Effective body shape

The box shape projected by the collision detection algorithm corresponds to the whole body minus the bottom part, defined by the width x step offset:



Probe Ground Method

Once you have moved the character on the ground the only thing left is to probe the ground in order to adapt the character to it. This process could involve changing the movement direction, check if the ground is dynamic and update the current information, climbing a step, check if the character is on unstable ground, check if there is even a ground at all.

This is the definition of stability for the character, regarding the ground:

Stable ground: slope angle <= max slope angle
Unstable ground: slope angle > max slope angle

These are all the posibles transitions:

- Stable ground -> Stable ground : The character will recalculate the grounded movement direction.
- Stable ground -> Unstable ground (Steep Ascending Slope) : The character will be push back, you will not see anything, it will seems that the character can't climb up.
- Stable ground -> Unstable ground (Steep Descending Slope) : The character will enter the not-grounded state.
- Stable ground -> No ground : The character will reset it ground data and enter the not-grounded state.
- Unstable ground -> Unstable ground : The character will be positioned onto the new slope and the slide action will continue.
- Unstable ground -> Stable ground : The character will recalculate the grounded movement direction.
- Unstable ground -> No ground : The character will reset it ground data and enter the not-grounded state.

Collision info

The “Character Motor” component acts as an interface to every class and component related to the core functionality. By calling the character motor you will be able to access all the public fields and getters inside. The available fields are:

Collision flags:

- isGrounded
- isHead
- isAgainstLeftWall
- isAgainstRightWall
- isCrushed

Horizontal obstacle information:

- wallObject
- wallSignedAngle
- wallAngle
- wallAngleSign

Ground information:

- slopeSignedAngle
- slopeAngle
- slopeAngleSign
- isOnSlope
- isOnRightSlope
- isOnLeftSlope
- isOnStableGround
- groundMovementDirection
- groundContactPoint
- groundLayer
- groundNormal
- groundObject

Interpolation

Support

If you read the features of the core you will notice that Kinematic 2D supports interpolated and non-interpolated movement, this means that you can choose to move and rotate using interpolated methods (like for example “MovePosition”) and non-interpolated methods (like for example Transform.Translate).

Motion mode

As mentioned before, the character would do the movement/rotation work at the end of the process of the character motor. The method use to perform this action is defined by the motion mode. The options are:

- Transform: The character will call the “Translate” and “RotateAround” methods.
- Rigidbody Non-interpolated: The character will assign the “rigidbody.position” and “rigidbody.rotation” fields.
- Rigidbody Interpolated: The character will call the “MovePosition” and “MoveRotation” methods.

Tutorials

1 - Implementing a basic “Debug” Character Controller

This step by step tutorial will show you how to design and setup a basic character controller (probably not the one you are going to use in your game) using the Kinematic2D scripts. This particular character is useful for debugging, i’ve use it myself while testing some behaviours of the character motor, collisions, physics, and so on.

Once you have assigned the character motor (already seen in the section “Creating a basic character”) follow these steps:

Create a C# script inside Unity, call it for example “DebugCharacterController2D”. The first thing to do is to include the Kinematic 2D code to your script, this is possible by using the namespace where the character motor is defined, which is *“Lightbug.Kinematic2D.Core”*

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

using Lightbug.Kinematic2D.Core;
```

Define the CharacterMotor reference and assign it to the CharacterMotor component of your Character (in this case i assume that the component is assigned to the character itself, but the character motor can be referenced by any script, just like every other component in Unity):

```
CharacterMotor m_characterMotor;

void Awake()
{
    m_characterMotor = GetComponent<CharacterMotor>();
}
```

Because we are going to need to move the character, let’s defined a speed field:

```
[SerializeField] float m_speed = 4f;
```

Create an update method (could be a FixedUpdate too).

Define the velocity vector based on the current input from the horizontal and vertical axes, and the speed:

```
Vector2 velocity =
(
    Input.GetAxisRaw("Horizontal") * Vector2.right +
    Input.GetAxisRaw("Vertical") * Vector2.up
).normalized * m_speed;
```

Finally, set the velocity of the Motor with the Input velocity:

```
m_characterMotor.SetVelocity( velocity );
```

When the game is running the character should be moving up/down/left/right. For the sake of this tutorial let's use the WASD keys (Horizontal and Vertical axes).

Homework:

- Try to move the character freely in the air.
- Try to add objects in the scene (make sure these objects have colliders and the obstacle and ground layer mask of the character include their layers).
- Make the character hit the ground from above, pressing the "S" key.
- Once the character is grounded use only the "A" and "D" keys and see how the character responds and adapts to the ground, whether is steps/slopes, stable vs unstable ground, etc.
- Force the character to the not-grounded state by pressing the "W" key while it's grounded.

Once you understand what is going on with this super easy example you will realize that the whole process is really simple to understand, there is a clear line between the character controller and Kinematic 2D character motor.

So, What's left? … A lot actually, this is the first step, for a average platformer game you will have to add gravity (accumulate a velocity to the previous velocity vector while the character is in the air), maybe set the velocity of the motor in a more smooth fashion (it could be done for example with a SmoothDamp function), define your custom states (for example with a Finite states machine), do whatever you want, here is where the fun part should begin, the Kinematic 2D Core part is done, the rest is up to you.

Good Luck!

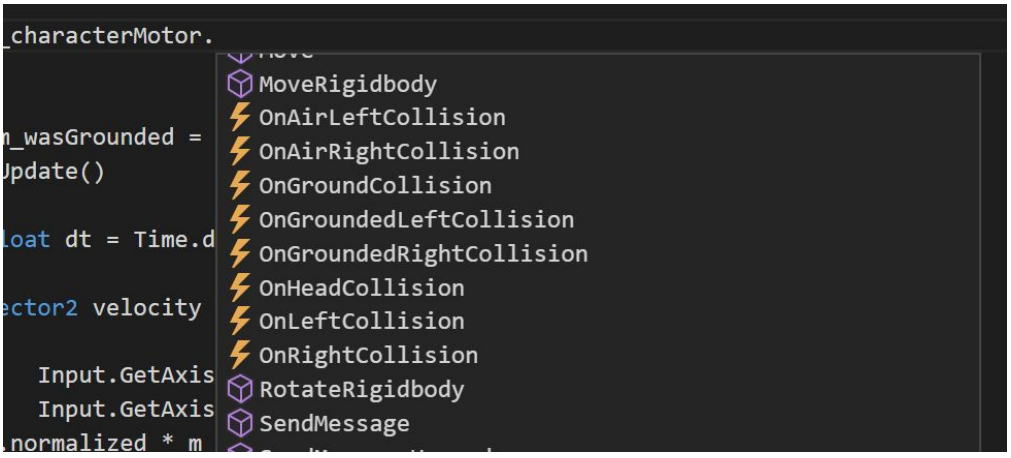
2 - Add listeners to the delegates events

Suppose that you need that your character do something everytime it hits the ground, you could easily write something like this:

```
if( m_characterMotor.isGrounded )
{
    if(!m_wasGrounded)
    {
        // Do the grounded stuff here

        m_wasGrounded = true;
    }
}
else
{
    m_wasGrounded = false;
}
```

Well, there is no need for that, because the character motor already has delegate events (not Unity events), these are “special methods”, once they are called (internally by the character motor) all the methods subscribed to them are called one by one.



The first need you need to do is subscribe to this events, it’s good practice to do this subscribe/unsubscribe thing in the OnEnable/OnDisable Messages from the Monobehaviour.

For example let's "subscribe/unsubscribe" to the *OnGroundCollision* event:

```
void OnEnable ()
{
    m_characterMotor.OnGroundCollision += OnGroundCollision;
}

void OnDisable ()
{
    m_characterMotor.OnGroundCollision -= OnGroundCollision;
}
```

See that the method passed to the delegate is "*OnGroundCollision*", I've chosen the same name just for commodity, although it doesn't matter the name of the method.

Inside the *OnGroundCollision* method you should put all the action that will be performed when the character hits the ground.

For this tutorial's sake let's just print a message:

```
void OnGroundCollision()
{
    Debug.Log("Ground!");
}
```

Every time the character enters to the grounded state a "Ground!" message should be printed.