USER MANUAL

Kinematic 2D 2.4.0

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

Versioning

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

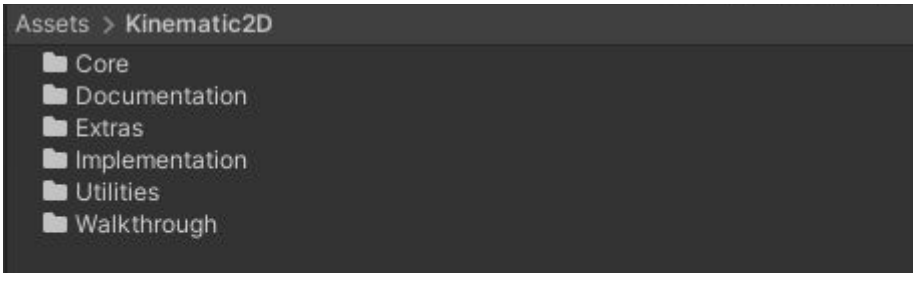
K2D(from version 2.1.1) is officially supported from version **2019.4.8f1 or higher**.

Unofficial supported versions

The core functionality of this package is prepared to work with **2018.3 or higher**.

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".

Setup

To be able to use the demo you'll need to load the **tags & layers** and **input** settings that come with the package.

Go to the layers inspector and load the "Kinematic2D_Layers" asset (placed at "*Kinematic2D/Documentation/Project Presets*"). Do the same for the inputs, go to the input manager inspector and load the "Kinematic2D_Inputs" asset.

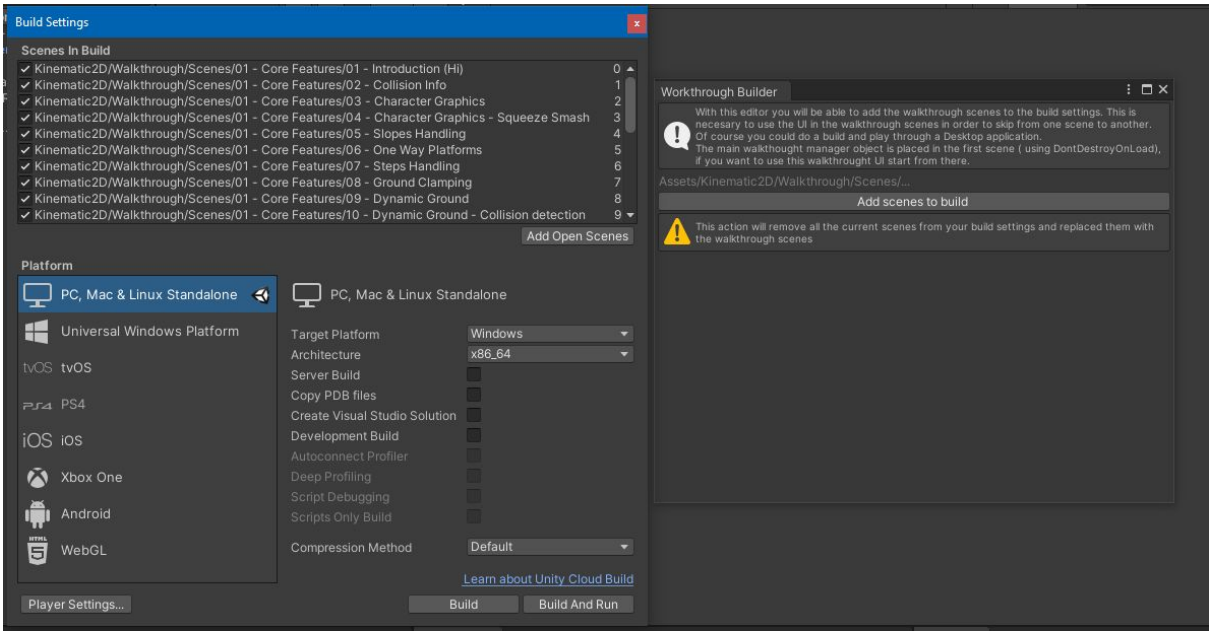
Walkthrough

In K2D is included a typical "Walkthrough section" with a bunch of small and concise scenes, created with the goal of introducing 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 does 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 accessed 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.

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 consists 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 these 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 these 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.

Setting up the scene (Adding the scene controller)

In order to be able to move characters and platforms, a scene controller is required. You can easily add one by drag and dropping a prefab called “Scene Controller” (Kinematic 2D/Core/Prefabs) into the scene. This prefab is just an empty object with a SceneController component attached to it.

Without this component everything will remain static .

Fixed step and Interpolation

Warning: In previous versions (<= 2.3.1) there were two extra update modes (Update and LateUpdate) and two extra motion modes (Transform and Rigidbody non-interpolated), however these modes have been deprecated, in favor of a fixed step update mode + interpolated movement.

The scene controller handles characters and kinematic platforms **using FixedUpdate exclusively**. A fixed step gives you the advantage of setting a fixed time (Project settings/Time). In order to get smooth results, the scene controller also interpolates every actor from the scene (using Unity's interpolation).

Creating a basic character

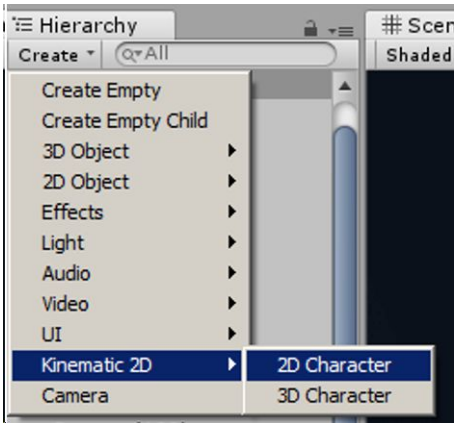
These instructions will help you to create a basic character that will not do anything, i know it does not sound very encouraging, but this is the first step and an important one.

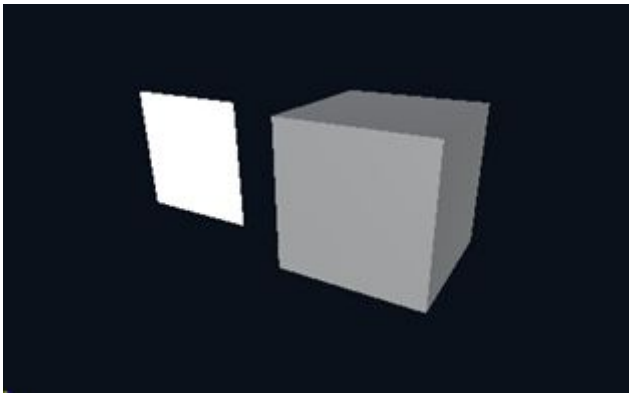
Steps:

1. Create an empty GameObject (The "Character").
2. Add a *CharacterBody*2D (2D Physics) or CharacterBody3D (3D Physics) to the character.
   IMPORTANT: There is no 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 a visual representation of your character.
5. Add to the visual object the "CharacterGraphics" component, this will allow 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".


Hierarchy
Create
All
Create Empty
Create Empty Child
3D Object
2D Object
Effects
Light
Audio
Video
UI
Kinematic 2D
Camera
2D Character
3D Character



Character body properties

For a detailed 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.

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.

NOTE: In previous versions the skin width was represented by a public variable (character body). Since 2.4.0 this is a constant defined in the CharacterConstant class. You can see this value in the character body inspector:

Step offset

The Step Offset determines the maximum step height the character can walk. The minimum value allowed for this is two times the skin width (a very low value).

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.

Movement methods

Velocity change

In order to move the character you can modify its velocity. To do so use the SetVelocity method. By default the velocity will be calculated as a **local velocity.** You can specify another parameter called “space” to indicate the coordinates used (“Self” or “World”).

For example:

1. SetVelocity(Vector2.right * 5f) will always make the character move towards the local right (transform.right), regardless of its orientation.
2. SetVelocity(Vector2.right * 5f , Space.World) will always make the character move towards the world right (Vector2.right), regardless of its orientation.

Teleport

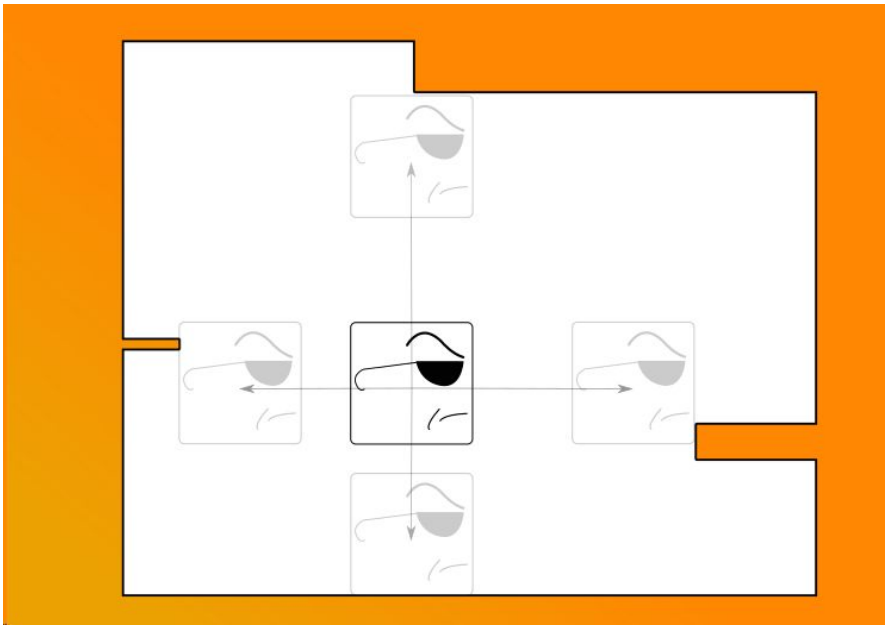
Teleporting is the same as changing the position/rotation. The reason this is useful is that the Teleport method also triggers an OnTeleport event. So, you can listen to this event and do your own things with the character or with anything else.

For example, you can make your camera script listen to this event. If the character teleported to somewhere else, then move the camera to the destination.

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).



The only way out of the not grounded state is (like the name implies) by getting in contact with the ground (grounded state).

Only vertical displacement will make the character go into a grounded state.

Effective body shape

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

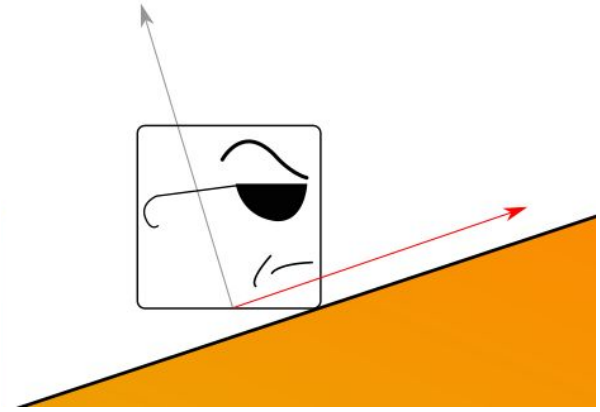
Grounded movement

The Grounded movement involves more steps than 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:

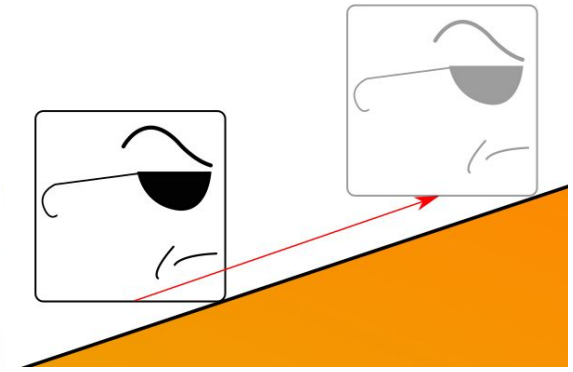


Look that a positive input x component doesn't match with the local right vector. For instance, the character could be climbing up a 45 degrees slope, however its vertical velocity will be zero.

TIP: To get the real displacement you can access the "PreviousDisplacement" property from the CharacterMotor.

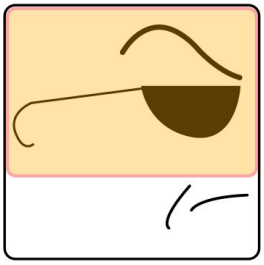
The grounded movement will always maintain the horizontal speed, it doesn't matter the slope the character is climbing. This is because the displacement is calculated by projecting the input velocity onto the ground.

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

This process could involve changing the movement direction, checking if the ground is dynamic (and updating the current information if necessary), climbing a step, checking if the character is unstable, etc.

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

Stable: slope angle <= max slope angle

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 properties inside. For more information about these properties please read their description (xml comments).

Basic tutorials

1 - Implementing a basic “Debug” Character Controller

This step by step tutorial will show you how to design and set up 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 used 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).

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.

Good Luck!

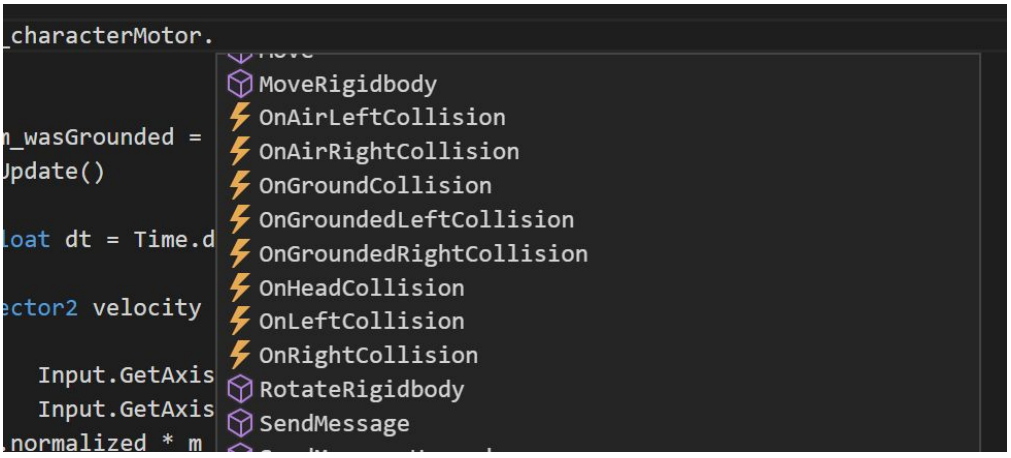
2 - Add listeners to the delegates events

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 the grounded state a "Ground!" message should be printed.