Retro Controller Manual

version 2.5

Topics

1. Introduction

Thanks for the purchase of the Retro Controller asset pack. If you like to start right away, jump to the next page.

The purpose behind this controller is to be a faithful recreation of the classic first-person gameplay found primarily in games like Quake, Half-Life/Counter-Strike. This include the smooth and fast movement contained in those games, along with techniques like bunnyhopping, airstrafing and surfing that the community developed along the years.

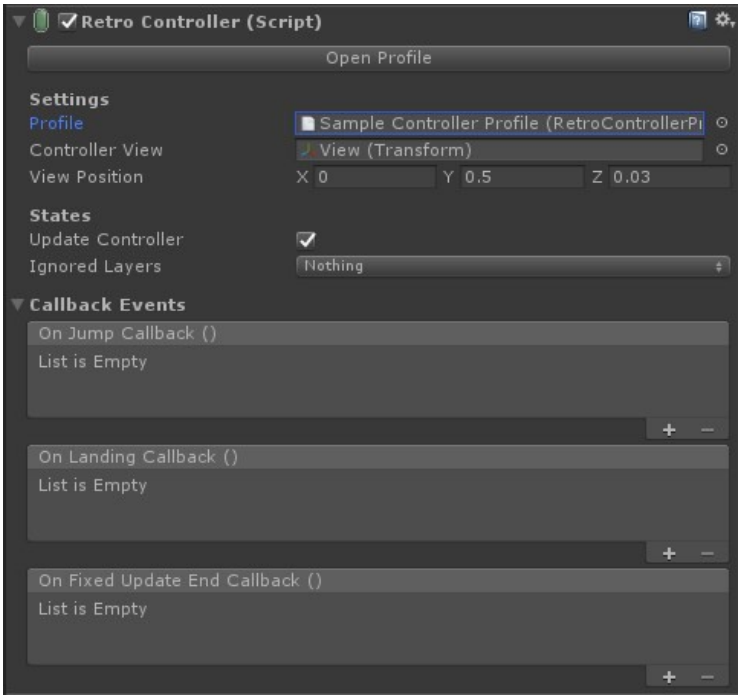
But if you like to make a more slow-paced FPS, the controller provides a whole range of options for you to customize how it should behave and therefore adapt it for your game design.

It’s really important to note that you have total access to the code that makes the controller run. So if you are a programmer, not only can you add features but also decide what’s useful, totally up to you and your team. There’s a special section in this manual covering more technical aspects of the controller.

2. The Retro Controller Component

The first thing you can do to start playing with the controller is to go to the **Retro Controller > Sample > Scenes** folder and open the **Tutorial 1** scene.

As you can see, the scene is a playground for you to experiment with the Retro Controller. Now got the **Hierarchy** panel and select the **Sample Controller** object.



On the **Inspector** panel you will see the component for the Retro Controller. There is a few options that is crucial for it to work properly.

The first one is the **Profile**, that will be explained more in depth in the **Retro Controller Profile** section. But basically, it's where settings like acceleration, friction and jump speed are. Without it, your controller won't run.

Next here is the **Controller View**. This is mostly used to set the water level and is assigned to the Game Object parent of the Main Camera. This will become clear on the section about the **Retro Controller View** component. The **View Position** serves as the starting local position for the **Controller View.**

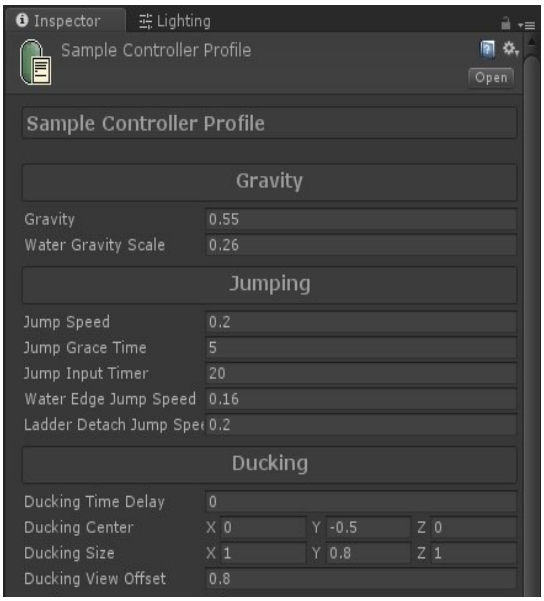
Under **States**, we have two options:

- **Update Controller:** you check this to *true* or *false* to determine if the controller will update each FixedUpdate call. This can be really useful if you want the controller to stop under some spell or aid you in pausing the game.
- **Ignored Layers:** best used at runtime, you can choose to ignore some layers depending on your gameplay. Keep in mind the collision layers are set at the **Collision Properties** in the Profile settings.

The **Callback Events** exposes certain behaviors that other scripts can easily subscribe to:

- **On Jump Callback**: this event is fired whenever the controller jumps, be through player or AI input. For example, you can use it to play a sound when the main character jumps.
- **On Landing Callback**: this is fired whenever the controller was on air and reaches the ground. Be careful, as this event can be fired multiple times when going down steep ramps, stairs or small elevations, as the controller tends to "fly" for a few frames. It's best to keep track on how long the player was in the air in case you wish to fire a lading sound or hurt the player for falling from someplace high, for example.
- **On Fixed Update End Callback:** since the controller relies on Unity's Physics engine, all the operations run inside a **FixedUpdate** loop. This callback is fired at the end of each iteration.

3. The Retro Controller Profile



Keeping the settings for the controller as a separate file brings a couple benefits:

- Being able to reuse configurations between multiple Game Objects using the Retro Controller.
- Make copies of the settings.
- Modify the settings during playtesting.
- Be independent from Game Objects or prefabs.

With that in mind, let's go through each section of the profile.

Gravity

Every game uses gravity and with this controller, it's no different. Here you can change settings on how it will be applied in different environments.

- **`Gravity`:** this one is pretty obvious, it controls the gravity force on the controller and it's oriented on the Y axis. A positive value means it will push the controller down (negative Y) and a positive will make it go up (positive Y), although the latter is not recommended.
- **`Water Gravity Scale`:** the previous value only applies when the controller is either on the air or on the ground. This will be applied only when it's submerged (more on that later) and it's *relative to the gravity value*. So if your **gravity** is set to **0.5** and the **water gravity scale** is set to **0.2**, the force result will be of **0.1**, since **the normal gravity is multiplied by the water gravity scale.**

Jumping

Here you can change how the controller will jump, along with some features that may improve the game feel.

- **`Jump Speed`:** this is the standard speed for jumping when the controller is on the ground and nowhere else.
- **`Jump Grace Time`:** some 2D platform games let the player jump after walking off the ground for a few frames. This is value is a timer measured in seconds and make the game more forgiving on precise jumps, hence the name **ledge forgiveness**. You can read more about that here.
- **`Jump Input Timer`:** this is a timer that helps with bunnyhopping, a technique that makes the player gain lots of speed and a crucial feature of this controller. It's an ability that may be too hard for some players, specially in games running at 60+ frames per second. With this

variable, you can make that easier.
Set the timer (in seconds) to a fair value and as soon as the player hits the jump button, the countdown will begin. When the controller is about to hit the floor, it will jump again and reset the counter, given the time is not over, preserving the velocity.
If you wanna learn about bunnyhopping, you can check out this video.

- **`Water Edge Jump Speed:`** when the controller is on a water surface (body underwater and view above the water) and reaches an edge, it automatically jumps so the player can get off the water.
- **`Ladder Detach Jump Speed:`** when the controller is on a ladder and the player sends a jump command, it will detach from the ladder and perform a jump **against the the ladder plane**.

Ducking

Gives the controller to ability to move through small areas in your game.

- **`Ducking Time Delay:`** when the player gives the *duck* command, you may wish the controller take some time to start ducking. Set this value to 0 to start ducking right away.
- **`Ducking Center:`** in order for the ducking feel natural, the center of the collider box needs to move down from the game object center.
- **`Ducking Size:`** it's common to change the collision box size so the player can move into small areas.
- **`Ducking View Offset`**:
- **`Ducking Lerp:`** enables or disables lerping of the collider values between standing and ducking.
- **`Ducking Lerp Speed:`** when this value is greater than 0, when the player ducks, the controller box doesn't change immediately. This value is ignored if **Ducking Lerp** is disabled.

Speed

Here you control how the speed will be limited or set in different states, from ground to water, from standing to ducking, among others.

- **`Max Ground Speed:`** standard speed limit, when the player is on the ground.
- **`Max Ground Sprint Speed:`** same as above but while the player is trying to run. If this value is lower than the standard, it can act as if the player is walking.
- **`Max Air Speed:`** speed limit when the player is on air. It can be broken if the **Air Control** option is set as ***Air Strafing***, by using the technique of same name. You can learn more about it here.

- **Max Water Speed:** speed limit while underwater.
- **Max Ducking Speed:** speed limit while ducking.
- **Max Vertical Speed Scale:** limits the value off the controller speed in the Y axis. This applies for both for going up or going down. So, for example, if this value is set to 2, the vertical velocity can't be less than -2 and can't be bigger than +2. This can avoid potential glitches on collision detection.
- **Ladder Speed:** when the controller is on a ladder the speed will be set to this fixed value. No acceleration, friction or gravity will be applied.
- **Max Air Control**: if the **Air Control** option is set to ***Air Strafing,*** this value will limit the amount of air control you can have normally in the air. That means you can move the controller in any direction while on the air.
  A high value will let the controller change it's velocity while in the air with total freedom.
  A low value may be good to let the player jump on elevations like boxes.
  A zero value will block any attempt at changing the velocity while in the air.

Acceleration

Acceleration is applied for the majority of the controller movements and defines how smooth the velocity transitions from standing still to full speed.

- **Ground Acceleration:** standard acceleration for when the controller is on the ground and standing.
- **Ducking Acceleration:** acceleration when the controller is on ground and ducking.
- **Air Acceleration:** this value determines how much the controller will accelerate in air but the behavior depends on the *Air Control value.*
- **Water Acceleration:** acceleration while underwater.
- **Air Control:** there is two possible options:
  - **Full:** complete air control, uses the *Air Acceleration* option to define how much it will be able to change it's velocity.
  - **Air Strafing:** allows the controller to change it's velocity only when performing the technique of same name, as explained before in this manual.

Friction

Any surface provides resistance and that's why things don't continue to move forever. Just basic physics law. The friction of the air is ignored in this controller.

- **`Ground Friction`:** basic friction on the ground. Doesn't matter if the character is walking, sprinting or ducking.
- **`Water Friction`:** friction while underwater.
- **`Fly Friction`:** although it was mentioned that air resistance is ignored, that is not true for flying characters. Their movement is similar to the ground, except gravity is ignored and they are able to go up and down. This kind of control is useful for things like Spectate mode in multiplayer games (when the player is just watching the game but not participating).

Controller Properties

This is where you control some core functionality of the controller and affects how it will act in the 3D space.

- **`Flying Character`:** as mentioned before, a flying character is not affected by gravity. It is also not affected by ladders, water and won't walk on stairs.
- **`Slope Angle Limit`:** this is the angle limit *relative to the Y axis*. What that means is an angle of 0 degrees represent a floor and an angle of 90 degrees represent a wall. So any surface above this value is considered a non-walkable surface and the controller will slide instead.
- **`Step Offset`:** steps are small elevations on the floor where the controller can collide. In order to simulate a human walking up on stairs, you can set this value in 3D space units to make the controller walk on these elevations.
- **`Center`:** this is the center of the box collider when it's standing. Usually it's centered on the game object position. Can change when the controller is ducking.
- **`Size`:** the size of the box collider when it's standing. Can change when the controller is ducking.

Collision Properties

In order for the controller to interact with the world, it must first identify some elements and what they are.

- **`Surface Layers`:** every game object has a layer. Here you can define which ones will the controller will collide with. Set them to *trigger* for water surfaces.
- **`Ground Check`:** the ground detection system uses this value to detect how far it should look for a ground. When the controller is on the ground, it ignores gravity.
- **`Water Tag`:** here you can set up which game objects are to be considered water. It must also have one of the *Surface Layers* and be set as trigger.
- **`Ladder Tag`:** mark the game object as a *Ladder* so the controller can climb on it.

- **`Platform Tag`:** usually in games, platforms act just like any solid object except for the fact it exerts force on other object by carrying them around. You can use this tag to implement your own platform solution and modify the controller's velocity when in contact with platforms.
- **`Depenetration`:** this is a special variable

Actions

Here you can find automated actions you can take on the profile.

- **`Duplicate`:** this lets you create a copy of the current profile into your project. It can be useful to test some parameters without losing your main profile.

Callbacks

Callbacks are **UnityEvents** used by the controller to invoke external functions. You can use this to get informed when certain actions take place within the controller:

- **`OnJumpCallback`**: called whenever the controller jumps.
- **`OnLandingCallback`**: called whenever the controller hits the ground. Be aware that while walking up or down on the stairs, this callback can be called multiple times.
  If you are using this to make landing sounds, there are a couple of approaches to make it better:
  - add a little delay between each time the landing sound plays.
  - play the sound after the player falls off a certain height or his `Velocity.y` is below a certain threshold.
- **`OnFixedUpdateEndCallback`**: since the controller deals with physics, it runs on `FixedUpdate` calls. You can use this callback to invoke actions that must take places after the collision solving process.

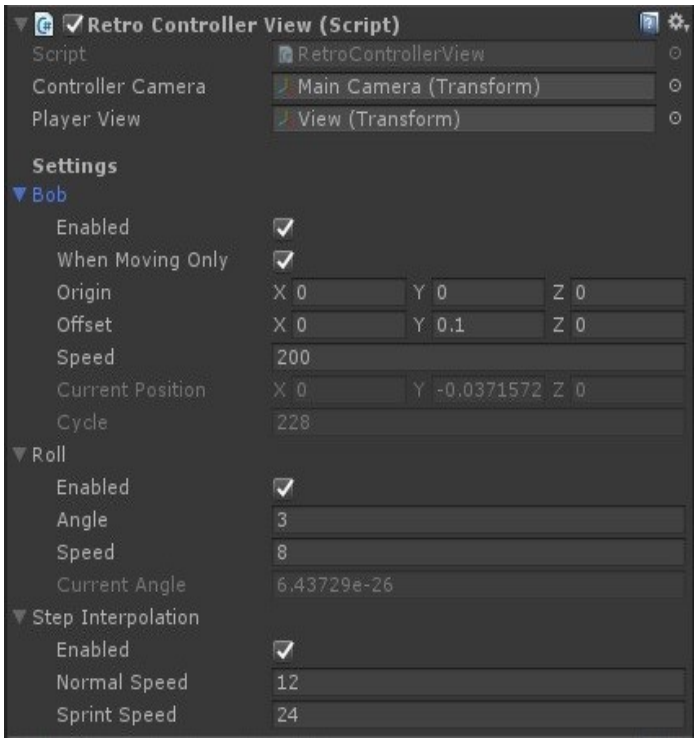
3. Input Configuration

In the Retro Controller, input is provided through a method called **`SetInput`**. It accepts the following parameters:

- **`forward [float]:`** accepts a value between **-1** (goes backward) to **1** (goes forward).
- **`strafe [float]:`** accepts a value between -1 (goes left) to 1 (goes right).
- **`swim [float]:`** accepts a value between -1 (swim down) and 1 (swim up). It's only used when on water.
- **`jump [bool]:`** accepts a true/false value.
  When on ground (has **IsGrounded** state) the controller will jump.
  If it's on the water, it will swim up.
- **`sprint [bool]:`** accepts a true/false value. Only works when standing on the ground.
  If receives a **true** value, the controller will sprint. It will move normally otherwise.
- **`duck [bool]:`** accepts a true/false value.
  When **true**, it will try to duck. It's gonna try to stand otherwise.

4. Camera Movements

The **`RetroController`** component comes with another component called RetroControllerView. It's an optional component that will give some camera movements.



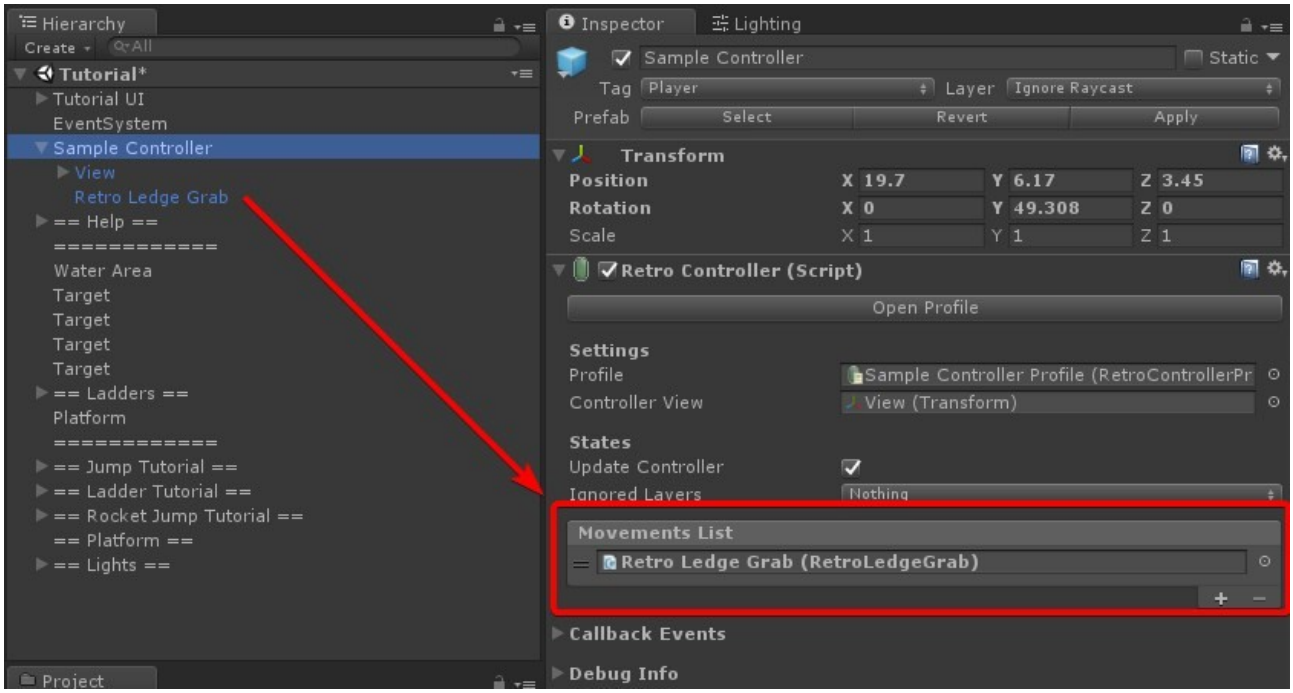
- **Bob:** or **head bobbing** makes the camera move between the defined **Offset**. Even though you can make it move left and right, it's best to keep just the Y value. Feels better with **When Moving Only**, meaning the camera will only move when the controller has velocity.
- **Roll:** the camera will roll in the Z-axis relative to it's camera. The screen will appear to rotate left and right accordingly to the direction the controller is strafing.
- **Step Interpolation:** this gives a smooth feeling while going up on small elevations or stairs. Without this effect, the player will feel the controller abruptly change it's position in the Y axis. If you wanna learn more about easing, check out this link.

5. Retro Movements

Retro Movements are a new way to extend the behaviors of the controller. While it covers most of the common movements from first person games, like swimming and crouching, it's not possible to take account of them all and games have varying gameplay demands. So, to make it easier for you to modify, this version comes with the **`RetroMovement`** class.

Here is a couple of steps to get you started:

- The first thing you should do is to **inherit** the class, since it's marked as abstract. Then you can start writing the code for your custom gameplay movement.
- Put your new component to any child GameObject of the your Player.
- Create a new entry on the **Movements List** and reference your component there. In the following example, we are using the **Retro Ledge Grab** (not available yet).



It's all done! The Retro Controller will automatically run your custom movement.

ATTENTION: MOVEMENTS ARE EXECUTED IN ORDER!

Make sure you place the ones with higher priority on the top of the list. Some movements might conflict with each other, for example, if you place `Retro Double Jump` before `Retro Ladder`.

Now, here is how the methods from the **`RetroMovement`** class works:

- `OnAwake`: this method is called inside the Awake method from the controller, so you can have a reference of the controller inside your custom movement class. You can also use it to initialize anything you might need.

- `DoMovement`: before doing any of the built-in movements, the controller checks for possible custom movements first since they have a higher priority. It's inside this class that you will need to check the conditions for the custom movement to run. If these conditions check, you need to return true to let the controller know the movement took place. Otherwise, you will return false, so the controller will check for another custom movement or eventually run the built in movements (ground, water, ladder or flying). You will need to call the CharacterMove method to move the character.

- `OnCharacterMove`: this method is called inside the **CharacterMovement** on the controller after the collision checks are done. There are multiple possibilities here, like setting flags or do additional physics checks. The upcoming samples will provide a more clear path on how to approach that.

The asset pack also comes packaged with 3 custom movements that makes use of the **`RetroMovement`** class. You can use them in your game or as a way to learn on how the feature works in practice.

Double Jump

As the name suggests, it lets the controller do another jump while in mid-air.

Example: https://vimeo.com/449887847

Crouch Slide

While running, press the crouch button and the controller will slide on the floor crouched.

Example: https://vimeo.com/449887824

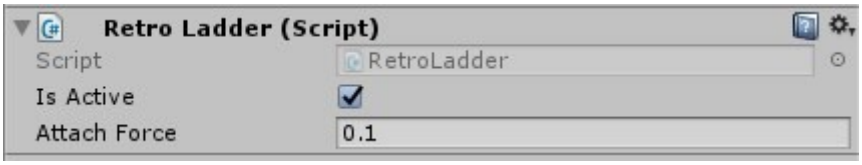
Ledge Grab

Let the controller grab ledges while in mid-air.

Example: https://vimeo.com/449887744

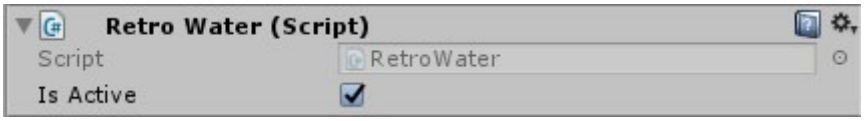
Ladders

Makes the controller climb on ladders. Ladders are any object with a Collider and a *Ladder* tag set. The `Retro Ladder` movement is the way you would prefer to add the functionality in your game.

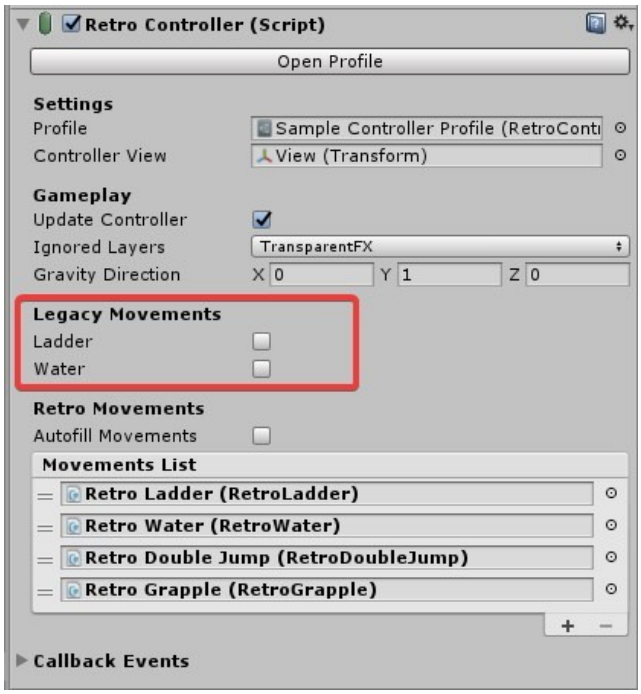


Water

Allows the player for swimming on determined areas. A water is any object with a Collider with `Is Trigger` enabled and with *Water* tag set. Using the `Retro Water` movement is the way you would prefer to add the functionality to your game.



6. Legacy Features



In previous version of the Retro Controller, climbing on ladders or swimming were hard-coded features. Now, they are marked as legacy features and are enabled by default for compatibility purposes. However, you should replace them with their respective Retro Movement versions as they won't be updated or supported anymore.

7. Extending the Controller

The RetroController solution aims to cover the most common behaviors found in the classic first-person games. But if you are a programmer familiar with C# and wanna add, remove or make any changes to the original code, the **`RetroController`** and **`RetroControllerView`** classes got you covered. Most, if not all the methods in the classes can be overridden, meaning that it's easy to replace small functionality without having to rewrite big parts of the code.

If you wanna learn how the controller works, open the files and look at the comments. If you have any additional doubts, don't be afraid to look for support.

8. Support

If you are looking for support, be either a problem you are having, code clarification or general questions about the controller, there is two ways you can contact me.

There is a Discord server and you can found the invitation link in the Asset Store page. Make sure to have the invoice code of your purchase and send me a DM with the code (username **epiplon**), so I can give you access to the RetroController support channel.

The second method is through e-mail at epiplonstudio@pm.me. Send the invoice and the question at the body of the message.

Please try to be clear as possible and, if you can, include screenshots of videos in case you have any issue.