Flare Engine Documentation

version

Two Bit Machines

July 30, 2022

Contents

Flare Engine Documentation

Introduction

Welcome! This is the documentation for Flare Engine. This tool provides an array of tools for creating 2d platformers much quicker, all without touching a line of code! It comes equipped with a streamlined user interface that makes the editor easy to work with. This will quickly get you going on creating your games!

If you are reading the autogenerated pdf version, please feel free to read the online version for a more readable experience:

twobitmachines.github.io/FlareEngineDocs

If you have any questions or feedback, please contact us at TwoBitMachinesDev@gmail.com

The Basics, Getting Started

At its core, like most custom implementations, the system uses raycasts. These raycasts will emit from the characters and detect the world, bypassing the physics engine. This will allow the user to have greater control over the game world. To get this foundation setup properly, each scene **must** contain a **World Manager** gameobject. Once this component is in the scene, it will automatically install the following Layers and Tags (if they don't already exist) to ensure smooth operation of the system:

Layers

- **World** - Ground, slopes, walls, ceilings. Use this layer on any gameobject that is a solid surface.
- **Platform** - Use this layer on any gameobject that is a Moving Platform.
- **Player** - Use this layer on the player gameobject.
- **Enemy** - Use this layer on any enemy gameobject.

Tags

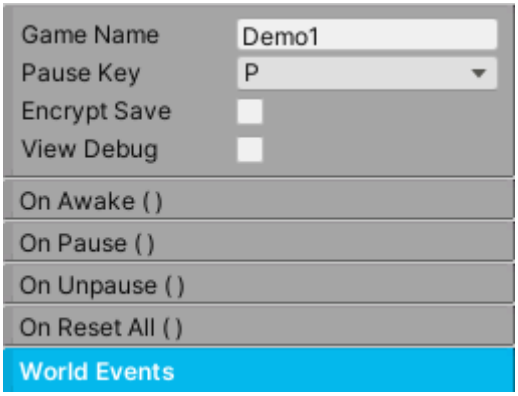
- **Friction** - Use this tag on any ground gameobject that is using the Friction component.
- **NoClimb** - If the player can climb walls, use this tag on any wall gameobject that is *not* meant to be climbable.

And that's basically it. That's all you need to get started. From here you can add a player, choose its abilities, design AI enemies, setup common systems such as Inventory or Dialogue, and implement world interactables to spice up your game.

World Manager

The **World Manager** is in charge of running the game scripts in order. It's also in charge of saving, pausing, resetting the game, and world events.

Each scene must contain exactly **one** World Manager for proper game function. Simply create a gameobject, add this component, and make sure it's always enabled. This gameobject should never be destroyed.



Property	
Game Name	The name of the game. Required for saving game data.
Encrypt Save	If enabled, this will encrypt the saved data.
Pause Key	A convenient button to toggle the pause state of the game.
View Debug	If enabled, this will show raycasts from projectiles and other game systems.
On Awake	A general purpose Unity Event called during Awake. This can be useful for initializing objects.
On Pause	The Unity Event invoked when Pause() is called.
On Unpause	The Unity Event invoked when Unpause() is called.
On Reset All	The Unity Event invoked when ResetAll() is called.

Method	
Pause()	This will pause the game and block player inputs.
Unpause()	This will unpause the game.
ResetAll()	This will reset all the relevant scripts in the game like the Player, AI, Interactables, etc.
Save()	This will save every Inventory in the scene as well as World Variables, including Health. This is automatically called during OnDisable();

Save

The engine implements JSON Serialization and basic encryption to save game data. Inventory and World Variables rely heavily on this. During development, disable Encrypt Save (so you can actually read the data) and check if your data persisted by going to Application.persistedDataPath. Typically, the path will look something like this:

%userprofile%\AppData\LocalLow\companyname\productname\TwoBitMachines\gameName

---

World Events

Send messages so world objects can react to important world events. These events are usually created by special events in the game, like the player dying. In response, any gameobject that is registered as a listener will be alerted and react accordingly.

To create a World Event, go to Add World Event, name the event, and click the add button. The World Event is actually a scriptable object (**WorldEventSO**) that can be found in the AssetsFolder/Events folder. Create as many as necessary.

Once the World Event has been created, it's going to need a triggering event and event listeners. **WorldEventTrigger** will be placed on the object that will trigger the event. To continue the example, when the player dies, its Health component will call WorldEventTrigger via Unity Event, and that will send an event to all the listeners. **WorldEventListener** is placed on any gameobject that needs to react to the event, like a UI element or an enemy AI.

World Event Trigger	
World Event	Choose the world event to trigger.
Trigger Event()	When this method is called, the specified world event will be triggered.

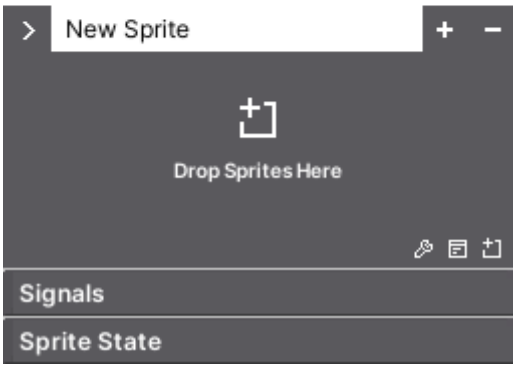
World Event Listener	
World Event	Choose the world event to react to.

On World Event	The event that wil be invoked when the world event is triggered.

Sprite Engine

The **SpriteEngine** component provides an extremely easy way for creating and storing sprites. The Sprite Engine removes the hassle of working with different systems and centralizes the power of creating sprites all in one place. Hence, there is no more need for creating animation objects or jumping from window to inspector to set things up. Everything is handled in the inspector. This system will even handle the logic for controlling the animation state of your character, **so there is no need to code**. The system really does aim to be a one stop solution for sprite animations.

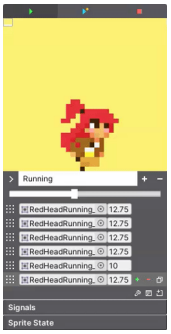
To get started, simply add the SpriteEngine component to any gameobject. A SpriteRenderer will be added automatically. You will then see this.



Property	
Play Buttons	Play the sprite in the inspector or in the scene.
Display Window	Displays the sprite. Drag and zoom the window. Right click to reset the window. Change the background color in the upper left corner.
New Sprite	Create, name, and remove sprites. Use the drop down menu to change between them. By default, the system creates one sprite called New Sprite. Rename it.
Drop Sprites Here	When a new sprite is created, drop the sprites here. Click this area to lock the inspector for convenience.
Wrench Icon	Add properties to each sprite to modify, like a Transform or Collider2D.
Options Icon	If necessary, set the sprite to loop once and set the OnLoopOnce event. Loop Start Index specifies where to start playing the sprite once it's already played once.
Drop Icon	Gives the option to replace all the current sprites.
Signals	Signals are used to control the animation state.

Sprite State	Create an animation state machine.

Once a sprite has been created, the sprite menu will appear. Rearrange, add, and delete sprites here. You can also test the play rate by adjusting the global speed slider. This will set the speed for all the sprites instantly. For customization, each sprite can also be applied an individual play speed.



Signals

Signals are just simple booleans but they're an integral part of the animation state machine. When a signal is set true, the state machine reads it and plays the corresponding sprite. That's it. For example, when a character is touching the ground, the system will set the onGround signal true. The Sprite Engine can then read this signal and play the standing animation.

The engine itself has the potential of setting many signals, and so for convenience, all the possible signals the engine can set are automatically provided (check player signals for the full list). All that is required is to click and enable the signal block you intend to use.

However, if you're not using Flare Engine, you will need to provide these signals to the Sprite Engine. This means you're in charge of setting the signals true. You will need a reference to Sprite Engine (using TwoBitMachines.TwoBitSprite) and set **tree.signal** which is a Dictionary<string, bool>. The string being the name of the signal and the boolean represents its active state. If you're using Flare Engine, there is no need to worry about this. The engine will be in charge of reading and resetting the signals.

And of course, it is possible to create your own signals by using the Create Signal field. These are mostly needed for creating attack animation signals.

Sprite State

The state machine that will be configured to set the animation state of a character. At its core, this system is only reading signals and setting the appropriate sprites accordingly.

Once all the signals and sprites have been created, click the add button on the bar to create a state. The animation state (depicted in orange) will have a list of all the available signals. Once the signal is chosen, click the drop down arrow to open the state and set the sprite (depicted in green) that should play if the signal goes true.

Once all the states are created, you must arrange them for priority from top to bottom. The system will check each state, starting from the top, and continue to the bottom until it finds a signal that is true. The system will then stop executing and set the appropriate sprite.

Each state can also have a sub-state in case there is a group of related signals that need to be organized. To create a sub-state, click the add button. States can be nested up to four levels with sub-states.

The example below shows a state machine with two states. The jumping state has two sub-states. If the jumping signal goes true, the system will check the signal velYUp, which means the velocity is positive. If it's true, the system will play the JumpingUp sprite. If VelYDown is true instead, the system will play the JumpingDown sprite.

The second state works the same way. You will also notice the blue state, which is responsible for flipping the sprite int the x or y direction. The main signal is set to alwaysTrue so that the system is always checking this state. The system then checks the signals and flips the sprite accordingly. The most common signals to check will be velXLeft and velXRight. If the x velocity is pointing in either of those directions, the system will flip the sprite in that direction.

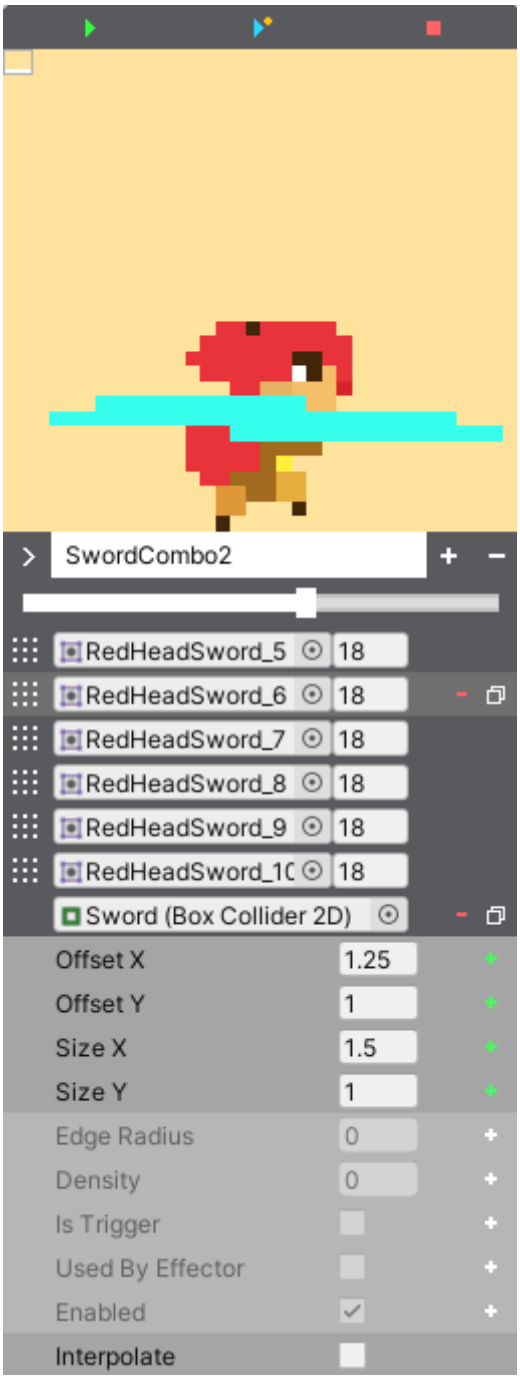
Important

All your sprites must be created facing to the right to work correctly with sprite flip.

Properties

This allows every sprite to have extra properties to modify and a per frame basis. Sometimes it's necessary to control a Transform or a Collider2D to work seamlessly with the animation. A very common scenario is to modify a Collider2D and change the size of its collision area to work with an attack animation.

Click the wrench icon and then select the type of property to work with. Once selected, the property will appear in the sprite menu. Set its reference and open it to begin modifying.



Tip

When working with properties, you can see the changes made to them in the scene.

In the example above, the animation depicts a sword attack. On the second frame, we can see the BoxCollider2D has its offset and size set to particular values. If the third frame is selected, the values for these settings can be the same or different, depending on what's necessary for the animation. The properties must be enabled by clicking the green plus button. You can also choose to interpolate these properties from frame to frame by enabling the interpolate toggle. Once enabled, the I button will appear next to each plus button. Click on it to enable interpolation for the specified property.

Tip

It is common for attack animations to loop only once. The OnLoopOnce event usually calls some method to let the system know the animation is complete.

Player

The player is the centerpiece of your game, and as such, the engine provides plenty out of the box customizable abilities. To get started, two components are required, a **BoxCollider2D** and the **Player** component. That's it. The Player class will handle the input, abilities, and world collision settings.

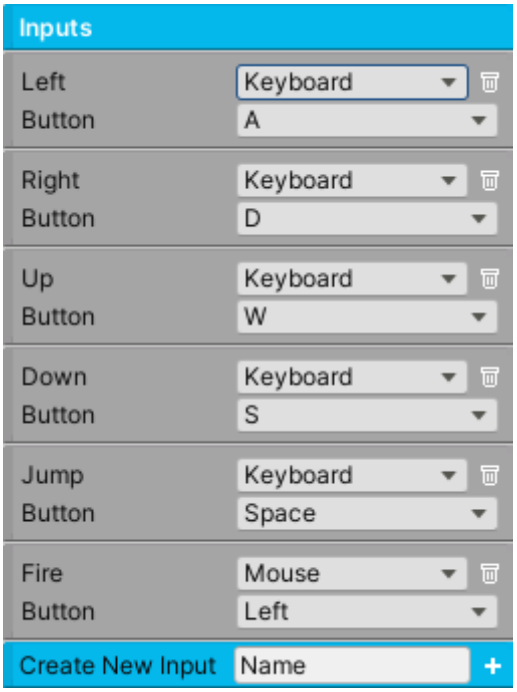
Method	
BlockInputs(bool value)	This will block player input if true.

Inputs

The engine implements a basic input system that can handle Keyboard and Mouse input directly. These inputs will be used to move the player and interact with the game world. And because each input is actually a scriptable object (**InputButtonSO**), it becomes very easy to read and set input values. So, if you wish to control the player by other means, it's entirely possible to control these inputs through method calls.

It's important to note you can change Keyboard and Mouse input keys during runtime, and the system will automatically save these changes using PlayerPrefs.

By default, the system will create common inputs for movement, jumping, and firing. To create an input of your own, click the add button on Create New Input. Once the input is created, an InputButtonSO is automatically created and placed in the AssetsFolder/Inputs folder for reference. The input you see in the inspector is directly linked to its InputButtonSO.



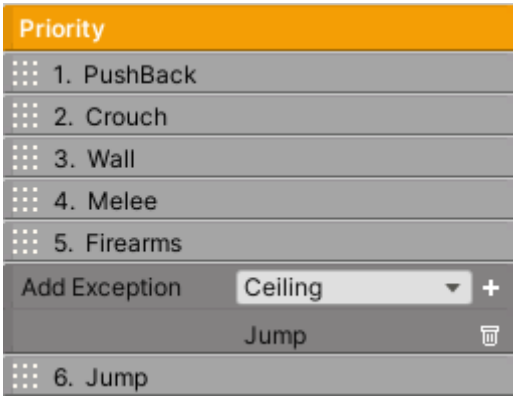
Inputs	

Name and Type	Pick the type of input, Keyboard or Mouse. If Event is chosen, this means the input will be set manually.
Button	Pick which key will be used by the Keyboard or Mouse.

InputButtonSO	Methods
ButtonPressed()	This will set button pressed as true.
ButtonHold()	This will set button hold as true.
ButtonReleased()	This will set button released as true and will set button hold as false.
OverrideKeyboardKey (KeyCode newKey)	Override the keyboard key.
OverrideMouseKey (int newKey)	Override the mouse key. 1 == Left. 2 == Right. 3 == Middle.

Priority

If the player has more than one ability enabled, you will have to assign their priority. This means that if two or more abilities become active at the same time, the ability with the highest priority will execute while the other abilities are cancelled. This is to prevent awkward situations. For example, let's say that when the player is climbing a wall, the player must not have the ability to shoot its firearm or else a shooting animation might commence. This will look weird if the shooting animation does not account for the wall holding. With the priority system, this can be prevented.



However, it's sometimes necessary for abilities to coexist. The system implements exceptions for these situations. Any ability can add exceptions, which allows the specified abilities to execute concurrently with it. For example, when the player shoots a firearm, if the Firearms ability does not include the Jump ability as an exception, the player will not be able to fire a projectile and jump at the same time. To remedy this, add the Jump exception to the Firearms priority.

To set a priority, simply drag the priority blocks over each another. **Number one is the highest priority**.

Tip

If using the PushBack ability, it's recommended to set it as the highest priority so when the player is damaged, it will disengage from whatever ability it is currently using, like climbing a wall.

Settings

Property	
Jump	The jump height and jump time dictate the value of gravity.
Rays	The number of raycasts that will extend horizontally and vertically from the character to detect the world. Keep this number low for efficiency.
Climb Slopes	If enabled, the character will be able to climb slopes no higher than the max slope.
Rotate To Slope	If enabled, the character will rotate according to the rotate rate to be perpendicular with the ground.
Rotate To Wall	If enabled, the character will be able to rotate from the ground to a wall below. For instance, the corner of a platform.
Rectify In Air	If enabled, and if the character is rotated and jumps, it will rotate to a squared position.
Check Corners	If enabled, the system will check if platform corners are going into the side of the character and prevent it. Note, if Climb Slopes is enabled, the system will automatically check for corners going through the bottom of the character.
Use High Jump	If enabled, the character will be able to use High Jump.
Use Bridges	If enabled, the character will be able to walk on bridges.
Use Moving Platforms	If enabled, the character will be able to interact with moving platforms.
Crushed By Platform	The Unity Event invoked when the character is crushed between two hard surfaces. This happens when the character is standing on or holding a moving platform and is moved into a hard surface, or when the character is pushed into a hard surface. This event typically means a character death, and the character's position needs to be reset, or else the character will move into a wall and create an ambiguous scenario. Call a World Effect with the dynamic Activate method.

Edge Collider 2D

When dealing with platforms that contain **EdgeCollider2D**, the character can jump through a platform if the jump button is being held. Otherwise, the system will interpret the platform as a ceiling. The character can jump down through a platform if the down button is being held. Of course, this can only work if the Jump and Down inputs exist.

Ability

Click the red button to add abilities into the system. Any ability can be paused by calling the Pause method from its component. For convenience, each ability can be paused in the inspector during development.

Creating An Ability

Create a unique ability if the engine doesn't already provide it. Follow the template below.

```
using UnityEngine;

namespace TwoBitMachines.FlareEngine.ThePlayer
{
    public class Ability : MonoBehaviour
    {
```

```
        // None of the methods below are required. Use them as necessary
        public override void Initialize (Player player)
        {
        // Initialize variables here. This is called in Awake.
        }

        public override void Reset (AbilityManager player)
        {
        //  Reset important variables here.
        }

        public override bool TurnOffAbility (AbilityManager player)
        {
        //  This is called when a higher priority ability overrides this one.
        //  Reset important variables. Usually the Reset method is called here.
        //  Return false if this ability is doing an important task that can't be cance
        //  For example, the crouch ability can't be cancelled unless the player has en
        //  head room to fully stand up. Return true if it can be cancelled.
            return true;
        }

        public override bool IsAbilityRequired (AbilityManager player, ref Vector2 velo
        {
        //  This is where the system checks if the ability has become active. Some cond
        //  will go true, usually a button is pressed, in which case return true.
            return false;
        }

        public override void ExecuteAbility (AbilityManager player, ref Vector2 velocit
        {
        //  Place the code that executes the ability here. This is only called if the a
        //  is active.
        }

        public override void EarlyExecute (AbilityManager player, ref Vector2 velocity)
        {
        //  This is always called before ExecuteAbility. Sometimes it's necessary to ex
        //  code before anything else. The ability itself shouldn't execute here. Howev
        //  if the ability is simple enough and always needs to run, execute the abilit
        //  The Ground ability functions this way. If doing this, just make sure the ab
        //  won't interfere with any other abilities.
        }

        public override void LateExecute (AbilityManager player, ref Vector2 velocity)
        {
        //  This is always called after ExecuteAbility. Firearm ability uses it to appl
        //  recoil to the player.
        }

        public override void PostCollisionExecute (AbilityManager player, Vector2 veloc
        {
        //  This is called after all the abilities and collision checks have executed.
        //  This velocity was the total velocity applied to the player during the frame
        }
    }
}

// The AbilityManager has many variables to be aware of.

// player.signals    -- set relevant signals: player.signals.Set ("signalName")
```

```
// player.world      -- read if player is onGround, onSlope, onMovingPlatform, etc
// player.inputs     -- read button inputs: player.inputs.Pressed("buttonName"), etc
// player.gravity    -- the value of gravity
// player.maxJumpVel -- the maximum jump force
// player.onSurface  -- set true if the player should be standing on a surface. Bridge uses
// player.jumpButton -- read if the jump buttons have been pressed
// player.hasJumped  -- set true if the ability made the player jump
// player.checkForAirJumps -- set true if the ability made the player jump
// player.playerDirection -- the direction of the player in the x-direction
```

Signals

During the life cycle of a character, the system, through the use of abilities and nodes, will set the signals below to true or false. For example, if a character is moving to the right, the system will read the character's velocity and set velXRight *true* if the velocity in the x direction is positive.

If using Sprite Engine, these signals are automatically read to control the animation state. There is no need to worry about anything else.

If you are not using Sprite Engine, then you can access the signals below through Character, which is the base class for player and AI. Use **Character.signals.signals** using namespace TwoBitMachines.FlareEngine. The signals class is a dictionary with a key-value pair of string and bool. Don't forget to reset these signals to false each frame or they will remain true.

- velX
- velXLeft
- velXRight
- velXZero
- velY
- velYUp
- velYDown
- velYZero
- onGround
- jumping
- airJump
- airGlide
- highJump
- windJump
- running
- hover
- ladderClimb
- ceilingClimb
- crouch
- crouchWalk
- friction
- sliding
- autoGround

- dashing
- dashX
- dashY
- dashDiagonal
- pushBack
- pushBackLeft
- pushBackRight
- inWater
- swimming
- floating
- wall
- wallLeft
- wallRight
- wallClimb
- wallHold
- wallSlide
- wallHang
- wallSlideJump
- wallCornerGrab
- recoil
- recoilLeft
- recoilRight
- recoilUp
- recoilDown
- recoilShake
- recoilSlide
- mouseDirectionLeft
- mouseDirectionRight
- meleeCombo
- zipline
- alwaysTrue
- alwaysFalse

Walk

Allow the player to move in the x direction. This ability is enabled by default.

Property	
Speed	The speed of the player in the x direction.
Smooth	A smoothing effect is applied to the speed every time the player starts or stops moving in the x direction. A value of one means there is no smoothing.
Impede Change	The resistance towards changing direction while the player is in the air. A value of zero means the player can instantly change direction in the air.

Run	If enabled, this will boost the Speed value.
Type	If Button is enabled, the user has to hold a button to run. If Time Threshold is enabled, the player has to walk on the ground for a specified time uninterrupted before starting to run.
Boost	Speed will be multiplied by Boost while the player is running.
Ease Into Run	If enabled, the player will ease into the running speed instead of accelerating instantly. Specify the duration of the ease time.
Ground Hit	The Unity Event invoked when the player initially hits the ground. Call a World Effect with the dynamic Activate method.
Not On Ground	The Unity Event invoked when the player initially leaves the ground. Call a World Effect with the dynamic Activate method.
Walking On Ground	The Unity Event invoked when the player is walking/running on ground. Call a World Effect with the dynamic Activate method.
On Direction Changed	The Unity Event invoked when the player changes direction on the x-axis. Call a World Effect with the dynamic Activate method.

Jump

The most fundamental ability of any platformer.

Note

Jump height and jump time are set in the Collision settings in order to calculate the force of gravity. However, this ability must still be enabled if the player is required to jump.

Property	
Button Trigger	Choose the button trigger for jumping.
Min Jump Height	If this value is greater than zero, the player will have a variable jump height, and min jump will be the lowest jump height possible.
Air Jumps	The number of extra jumps the player can perform in the air. The second field will scale the jump force. If using a specific air jump sprite, take advantage of Sprite Engine's Loop Start Index.
Air Glide	If holding the glide button, the player will gently glide down instead of falling when jumping. Air Glide can only occur after all the air jumps (if any) have occurred. The glide value must be between 0 and 1f.
Jump From Fall	If the player walks off a platform (without jumping), the player can still execute Air Jumps or an Air Glide if enabled.

Event	
Jump	The Unity Event invoked when the player jumps. Call a World Effect with the dynamic Activate method.
Air Jump	The Unity Event invoked when the player air jumps. Call a World Effect with the dynamic Activate method.

Important

If an ability already contains a jump force, do not add the Jump ability as an exception to it. For instance, the Wall ability contains a few jumping options that it will execute internally. The Jump ability is only geared for jumping on ground.

Signals: airGlide, airJump, airJump1, airJump2, airJump3

Wall

The player can slide, climb, corner hang, corner grab, and jump on walls. If a particular wall is not meant to be climbable, add a Unity tag “NoClimb” to this gameobject.

Slide

Property	
Slide When	If Pressing Input is enabled, the player will slide on a wall only if it’s pushing against it. If Automatic is enabled, the player will latch onto the wall and slide automatically.
Slide Friction	How quickly the player slides down. The slide friction can be applied as a velocity or an acceleration.
Slide Timer	If enabled, limit how long the player can slide before falling.
Jump Up	The force that pushes the player up the wall. The X force will push the player away and then towards the wall.
Jump Away	The force that pushes the player away from the wall.
Jump Limit	Limits the amount of wall jumps before the player falls down.
On Slide	The Unity Event invoked when the player is sliding down a wall. This event will pass the player’s position.

Signals: wall, wallLeft, wallRight, wallSlide, wallSlideJump

Climb

Property	
Buttons	If Button is enabled, specify the up and down climb buttons. If Player Direction is enabled, the player will climb the wall in the direction of its horizontal movement. If None is enabled, the player will only be able to hold the wall.
Hold Wall	If Automatic is enabled, the player will hold the wall automatically. If Hold Button is enabled, specify the button that must be pressed in order to hold the wall.
Speed	How quickly the player climbs the up and down the wall.
Climb Limit	If enabled, limit how long the player can climb the wall before falling. If Fall amount is greater than zero, the player will hold the wall for this time period before finally falling. During the Fall period, the player will not be able to climb the wall.
Slide Down	If enabled, the player will slide down the wall instead of climbing down.
Jump Up	The force that pushes the player up the wall.
Jump Away	The force that pushes the player away from the wall.
Jump Limit	Limits the amount of wall jumps before the player falls down.
On Climb	The Unity Event invoked when the player is climbing a wall. This event will pass the player’s position.

Signals: wall, wallLeft, wallRight, wallHold, wallClimb, wallSlide

Note

Once the player is near the top of a wall, it will automatically jump on top of the platform. If Corner Grab is enabled, this setting will be ignored.

Corner Hang

Property	
Offset	Determines the point at which the Corner Hang becomes active. A value of 0.5f will check for a corner 0.5f units below the player's head.
Exit Hang	When to exit a Corner Hang. If Button is enabled, the up and down buttons will unlatch the player from the corner. If Player Direction is enabled, the player's movement will unlatch it from the corner. If None is enabled, only a jump action can unlatch a Corner Hang.

Signals: wall, wallLeft, wallRight, wallHang

Corner Grab

Property	
Type	How will the player grab the corner? If Jump is enabled, the player will simply hold the corner until the jump button is pressed. If Pull Up or Pull Up Auto is enabled, a corner climb animation will play. If Pull Up Auto is enabled, the player will start the corner climb animation immediately. Press the blue button to add each sprite in the animation.
Exit Grab	Once the Corner Grab state has been entered, the player can exit this state either by the specified button press or by player movement.
Animation Time	The total duration of the corner climb animation.
Sprite	A sprite in the corner climb animation.
Displacement	The system will control the position of the player while climbing the corner. The displacement will offset the player according to the sprite that is currently playing. This might take a little trial and error to get correct.

Signals: wall, wallLeft, wallRight, wallHold, wallCornerGrab

Important

If Corner Grab is playing an animation, it will use the Sprite Renderer attached to the player. If Sprite Engine is being used, it will be paused, allowing the animation to play freely. If using another system for playing sprites, the signal wallCornerGrab will be set True, which will let you know when to pause this other system.

Ceiling

The player will latch onto a ceiling and move in the x direction. This is not meant to work on sloped ceilings.

Property	
Friction	The friction applied to the player's x direction while moving and holding a ceiling.

Auto Grip	If enabled, the player will automatically latch onto a ceiling upon contact. To dismount, the jump button must be pressed. If not enabled, the user has to hold the jump button to ceiling climb.
Edge Jump	If enabled and if the ceiling is an EdgeCollider2D, the player will jump up through the platform. Scale the jump force if necessary.

Tip

If the ceiling is an EdgeCollider2D, auto grip will need to be enabled for proper function.

Crouch

Let the player crouch and crawl. This will modify the size of the BoxCollider2D.

Property	
Button	If pressed, the player will crouch.
Crouch Height	The new height of the player. This will modify the vertical size of the BoxCollider2D.
Crawl Speed	If enabled, the player will be able to crawl. Specify the friction applied to the player's x direction while crawling.
Crouch Jump	If enabled, the player will be able to jump while crouching and crawling. Specify how much to scale the jump force.
High Jump	If enabled, the player will perform a high jump from the crouch state.
Jump Boost	The player's jump force will be multiplied by this number during the High Jump.
Charge Time	The total crouch time that has to elapse before triggering a High Jump. The player's x velocity will have to be zero during this time.
On High Jump	The Unity Event invoked when a High Jump executes.

Dash

Increase the speed of the player to quickly cover distance.

Property	
Buttons	The buttons that need to be tapped in order to trigger a dash.
Dash Direction	If Horizontal Axis is enabled, the dash will occur along the x axis. In this state, only the left and right buttons are used. It is also possible to use only one button and leave the other empty. If Multi Directional is enabled, all the buttons that are set will be utilized to move the player in one of eight directions along the x and y axis.
Button Taps	If Single Tap is enabled, pressing the button only once will trigger a dash. If Double Tap is enabled, pressing the button twice is required to trigger a dash.
Tap Threshold	If Double Tap is enabled, the threshold is the time interval in which the double tap must occur for the dash to trigger successfully.
Duration	If Instant is enabled, the player will traverse the dash distance in one frame. If Incremental is enabled, the player will traverse the dash distance according to the dash time.
Dash Time	The time it will take to traverse the dash distance;
Dash Distance	The total distance traversed while dashing.
Cool Down	The time interval before the next dash can be triggered.
Crouch	If enabled, the player will change to a lower height while dashing. If the player is below a platform when the dash completes, the crouch signal will be set.

Can Take Damage	If disabled, the player will not take damage while dashing. This assumes the player has a Health component.
On Ground Only	If enabled, the player must be on the ground in order to begin a dash.
Nullify Gravity	If enabled, the force of gravity will not affect a dash.

Signals: dashing, dashX, dashY, dashDiagonal, crouch

Hover

Escape gravity by letting the player hover in the air.

Property	
Thrust	The forced used to propel the player upward. This force will be proportional to the jump force.
Maintain	The tendency for the player to remain in the air. A value of one will prevent the player from descending downward, unless the descend button is pressed.
Thrust Button	Press this button to create thrust.
Descend	The force that will drive the player downward. The descend button is optional. If it's not used, the player will descend on its own, according to the maintain value.
Descend Button	Press this button to create downward thrust.
Exit	If On Ground Hit is enabled, the player will exit the hover state when the player touches the ground. If Button is enabled, the player will exit the hover state when the specified button is pressed.
Air Friction X	The air resistance applied to the player while hovering in the x direction.
On Thrust	The Unity Event invoked when the thrust button is pressed.
On Descend	The Unity Event invoked when the descend button is pressed.

Signals: hover

Swim

Allow the player to swim or float on any body of water. The body of water will determine if the player either floats or swims. If floating, the player will remain above the water line. If swimming, the player will swim inside the body of water.

Note

The player must have the Swim ability enabled to interact with water.

Property	
Spring	If floating, when the player enters the water, it will oscillate on the water line before coming to a rest. This force dictates how quickly the oscillations occur.
Damping	How quickly the spring force dissipates.
Weight	How quickly the player sinks while swimming.
Water Impact	The force exerted on the water upon entry. The force exerted while the player moves in the water will be proportional to this value and the player's velocity.
Water Friction X	Water resistance applied to the player in the x direction.
Water Friction Y	Water resistance applied to the player in the y direction.
Jump	The force used to jump out of the water.

Switch Button	If water Switch Type is set to Yes, holding this button will transition the player from a floating state to a swimming state. To return to a floating state, the player must reach the top of the water.
On Enter Water	On water entry, a Unity Event containing the entry position is invoked. This could be useful for adding particle effects.
On Exit Water	On water exit, a Unity Event containing the exit position is invoked.

Note

A prefab called Bubbles emits bubble particles as the player swims. Make it a child of the Player. Feel free to change the particle settings, too!

Signals: inWater, swimming, floating

Ground

The player can interact with ground that is modified by the Friction interactable. This contains no settings.

Signals: friction, sliding, autoGround

Firearms

All firearms must be created through the Firearm class. This class is required to let the system know the player is using firearms.

Event	
OnCancel	The event invoked when this ability is cancelled by a higher priority ability.

Melee

Register the melee attacks so they can be properly executed. Unlike firearms, only one melee attack can be active at at a time. The melee attack that will be executed will always be the first one in the list.

Property	Method
CompleteAttack	Call this when the attack animation is complete to stop the melee attack.
ChangeMeleeAttack (string meleeName)	If more than one melee attack is registered, call this method to change to a different one by moving it to the top of the list.
MeleeAttackIsActive	Returns true if the first melee attack is active.

Ladder

The player can climb ladders.

Property	
Latch	If Automatic is enabled, the player will automatically latch to the ladder on contact, provided the player has a negative y velocity and a zero x velocity. If Enter Button is enabled, specify the button that must be pressed in order for the player to latch onto the ladder.
Climb	If Manual is enabled, specify the buttons (Up, Down) for climbing the ladder. If Automatic is enabled, the player will climb the ladder automatically.
Climb Speed	How quickly the player climbs the ladder.
Stand On Top	If enabled, the player can stand on top of the ladder.

Align To Center	If enabled, the player's x position will align with the center of the ladder.

Signals: ladderClimb

Rope

The player can interact with ropes.

Property	
Swing Strength	The force added to the swing motion.
Jump	If latched, the force used to jump away from the rope.

Push Back

If the player is damaged, apply a push back velocity in the relevant direction. This ability works in tandem with the Health component. When the Health value changes, use its OnValueChanged Unity Event and connect it to the ActivatePushBack method (found on the Push Back component). That's it. This event will send two values, damage and direction. If damage is negative, Push Back will commence.

Note

Typically, Push Back is set to the highest priority so it can override all other abilities. For example, if the player is climbing a wall, the player will fall down when Push Back is triggered.

Important

It's possible for the Crouch ability to override any ability if the player can't fully stand up due to a low ceiling. If this is the case, Push Back will not be triggered, even if it has a higher priority. To prevent this, add Crouch as an exception to Push Back so that both abilities can execute at the same time.

Property	
Distance	The total displacement in the x direction.
Jump Force	The force applied in the y direction. This force is applied only once.
Time	The total duration of the push back. The x velocity will be overridden during this time period.
Sprite Renderer And Material	The player will be applied a damage flash during push back. Set the Sprite Renderer of the player. If the material is set, it will be applied to the Sprite Renderer; otherwise the color of the sprite will be tinted. You can use the Material Hurt that comes with the system.
Color	The color of the flash.
Flash	The number of times the flash occurs during push back.

Tip

Use the signals to maintain the same direction of the Sprite during push back.

Important

If the damage flash is applied, the inspector will create a lag spike when the material is being swapped. To prevent this, simply select another gameobject that is not the player.

Signals: pushBack, pushBackLeft, pushBackRight

Ziplining

Enable this to allow the player to interact with Ziplines.

Property	
Zip Speed	The force applied to the x velocity of the player while ziplining. A value below 1 will feel like friction, and a value above 1 will be a speed boost.
Jump Force	The exit jump force.
Y Offset	The offset applied to the player's y position. This can be useful to align the animation.

Signals: zipline

Jump On Enemy

Jump on an enemy to deal damage.

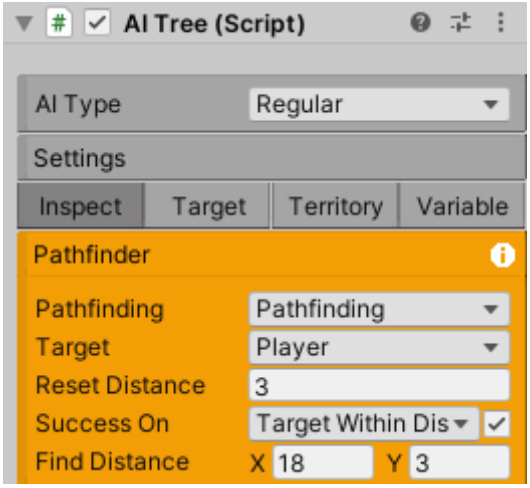
Property	
Layer	The layer the enemy belongs to.
Damage	The damage amount dealt to the enemy.
Bounce Force	The jump force applied to the player after contact with the enemy.

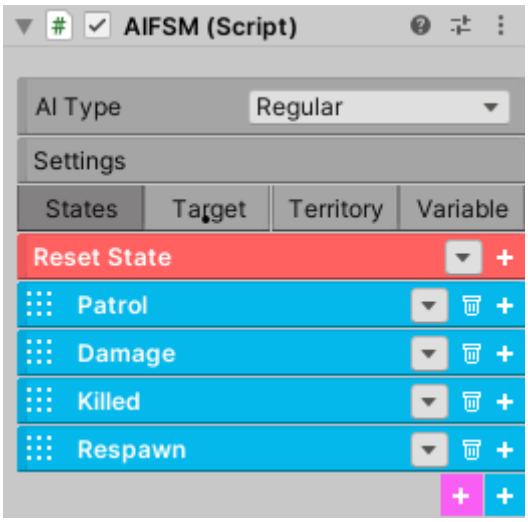
AI

Create truly complex AI using Behavior Trees or Finite State Machines, all without using a single line of code. The AI system is universal so that behavior can be applied to all aspects of the game. Besides creating enemy AI, you can program doors, moving platforms, design a save system, or create any utility that requires complex logic.

The two classes that implement AI is **AITree** and **AIFSM**. Before creating an AI, you should have an idea on how complex the behavior is. Typically, use a FSM for simple AI as this will generally be faster to create. Behavior Trees should be used for AI that would otherwise have too many states to handle in a FSM.

Both AI systems utilize nodes to create behavior. The system itself comes with over forty nodes, allowing for a lot of functionality for creating practical AI out of the box. However, there may come a time when none of the nodes in the system provide the desired behavior, in which case, you can always create a node to fulfill this need. Below is an example of how these classes look in the inspector.





Behavior Tree

Behavior Trees offer an alternative to finite state machines. They move away from the concept of state, and instead rely on hierarchical nodes to execute actions. This gives the developer flexibility for adding behavior into the tree and the ability to create truly complex designs without having to worry about state jumping logic.

How it works: the tree will always execute from left to right and from top to bottom. That is, nodes that are the most left and highest on the tree will have priority over lower nodes. Once a higher priority node is no longer active, the tree will evaluate the next highest node, and so on.

It is important to note the system will *not* reevaluate the entire tree each frame. This is a good thing, since we want the tree to be efficient and to only execute the current active node. To allow the tree to check for inactive priority nodes, the system implements interrupt checks. Interrupt checks must be enabled and are found in each Composite node.

Nodes

Each node can have one or more of three states:

- Success: the node is done running and succeeded.
- Failure: the node is done running and failed.
- Running: the node is still running.

And there are four node types:

Conditional Node

Returns Success or Failure. Its main purpose is to test a condition in the game world, like checking if the player is nearby, so the tree can execute an Action node in response.

Action Node

Modifies the AI by changing its state in the game world. For example, apply a velocity to the AI so that it moves towards the player. These nodes typically return Running.

Composite Node

This node runs a list of child nodes. Composite nodes come in different types, and the type will determine how these child nodes are executed. For example, a Sequence will run its child nodes in sequence until it finds one that returns Failure. A Parallel will run all its children at the same time.

Decorator Node

This node can only have one child node, and it will modify the output or behavior of this child node. For example, sometimes it may be necessary to execute a child node after a time delay or perhaps to invert the output of the child node.

Let's look at a very basic example for a patrolling AI. A Sequence has two child nodes. The first node, a Condition, will check if the player is within distance. The second node, an Action, will move the AI towards the player. Only if the Condition returns Success will the Sequence execute the second node. If the Condition returns Failure, the AI will not chase the player.

Blackboard data

The Behavior Tree also implements blackboard data, which is data that can be shared between nodes. Blackboard data is separated into three categories, and each category exists within its own tab.

Target are objects or points the AI will use for detection or following purposes. Target player is the most common, but of course targets can be transforms or vector points or even the mouse.

Territory is a rectangular area in the game world used for detection. For example, the Find Target node will alert the AI if the player has entered the specified territory. Territories can be dragged and resized in the scene. The other two territories are for pathfinding.

Variable is data that needs to be shared and *modified* by the nodes. Sometimes it is necessary for two nodes to modify the same list, in which case you might create a TransformListVariable. Of course, the system implements all the common data types for variables.

It's important to highlight two special variables. If using a Behavior Tree, each AI will automatically contain a **Bool Variable** called Reset. You cannot delete this. This boolean will be set true when the **WorldManager** triggers a game reset. Somewhere in the tree, usually at the highest priority, the Variable Logic node will be reading the reset boolean, and when the boolean goes true, it will reset the AI (maybe resetting its position or health). This boolean should then be set false using the Set Value node. Of course this is optional. If the AI doesn't need to be reset, the reset bool can be completely ignored. You can also trigger this event manually by calling the ResetAI method on the AI class.

The **Tree Variable** is extremely important for communication. If two different AI systems need to communicate with each other, then simply drop the reference of an AI class into this variable. All the blackboard data of an AI will become available to the AI implementing the Tree Variable. This means the two AI systems can read and modify the same variables.

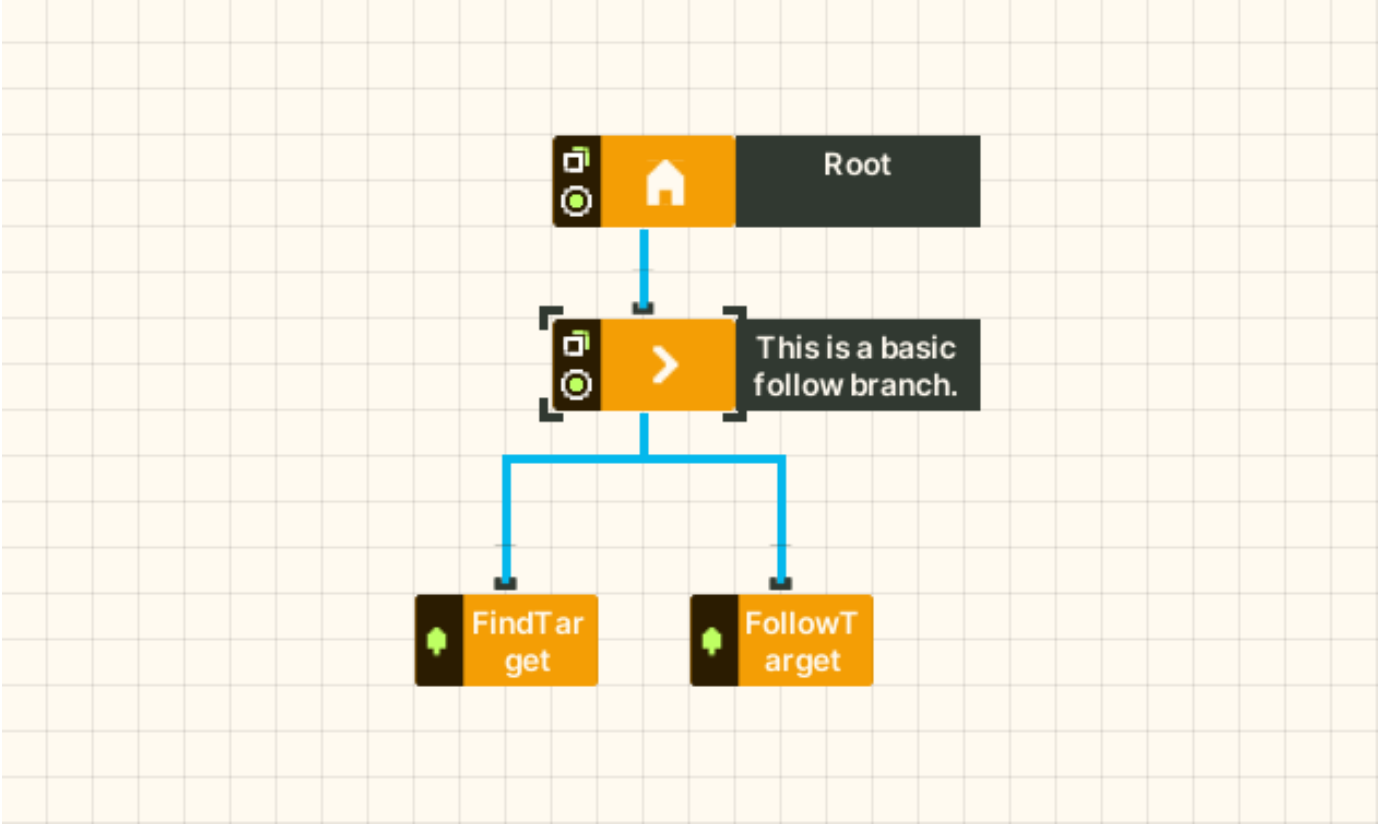
Each blackboard data you create must be given a unique name. This data, once created, will become readily available to any node that requires it. For example, the Follow Target node has a public member called target that is a **Blackboard** type. When a field is of this type, in the inspector, simply click on it and choose an option from the available drop-down menu. That's it.

If you are creating a node from scratch and need a blackboard data, simply add a **Blackboard** type as a public member and name it.

Tip

If any blackboard data is renamed, make sure to click on any node using it to refresh the references.

Node Editor Window



If using AITree, a node editor will be required to create the nodes and to configure the node hierarchy. In Unity, go to Window/BehaviorTreeEditor to open the node editor window. The window can be zoomed and dragged. Next, select a gameobject and add the AITree component. You will then see a single node appear in the editor. This is Root, which is the entry point into the Behavior Tree and all nodes and branches belong to it.

Right click anywhere in the node editor and the node context menu will appear. Select a node to create and it will too appear in the node editor. Go to the Inspector of the AITree. If you select the newly created node, the Inspect tab will display this node. This is where you will be changing the public fields of the node. Thus, you will be working with both the node editor window and the inspector to create a Behavior Tree.

Once you have created the necessary nodes, connect them. Each node that can establish a connection (Root, Composites, Decorators) will have a circle on the left side of the node. Click this circle then go to the node you wish to connect and click on top of the node (where the black mark is). If the connection worked, a blue line connecting the two nodes will appear.

As a reminder, the Behavior Tree is executed from left to right and from top to bottom. If a node has two children, the child node that is on the left will have priority. If you click and drag the other child node, changing its position to the left of the first node, the system will automatically establish this node as having a higher priority.

If you right click on any node, another context menu will appear. You can delete the node itself or the entire branch belonging to the node. You can duplicate the node. You can also add as many notes as necessary to the node. The note, if clicked, will appear in the inspector where you can type the necessary information. These notes can be moved and resized for convenience.

You can also create branch templates. For example, maybe you have a simple patrol branch that you wish to recreate in other Behavior Trees, well you can save this as a template to recreate later whenever you wish. These templates will be available in the node context menu. For now, the only way to name these templates is by creating a note. Whatever is typed in that note will become the name of the template. You can then simply delete the note if it's not necessary anymore.

When the game enters play mode, the active nodes and connections will turn green for debugging.

Interrupts

A Behavior Tree will typically have lower and higher priority branches. Once a lower priority branch is executing, by default the system will no longer check if a higher priority branch needs to be executed. This is to prevent the system from executing the tree from the very beginning each frame. To get around this issue, **Interrupts** are implemented to allow higher priority branches to interrupt lower priority branches.

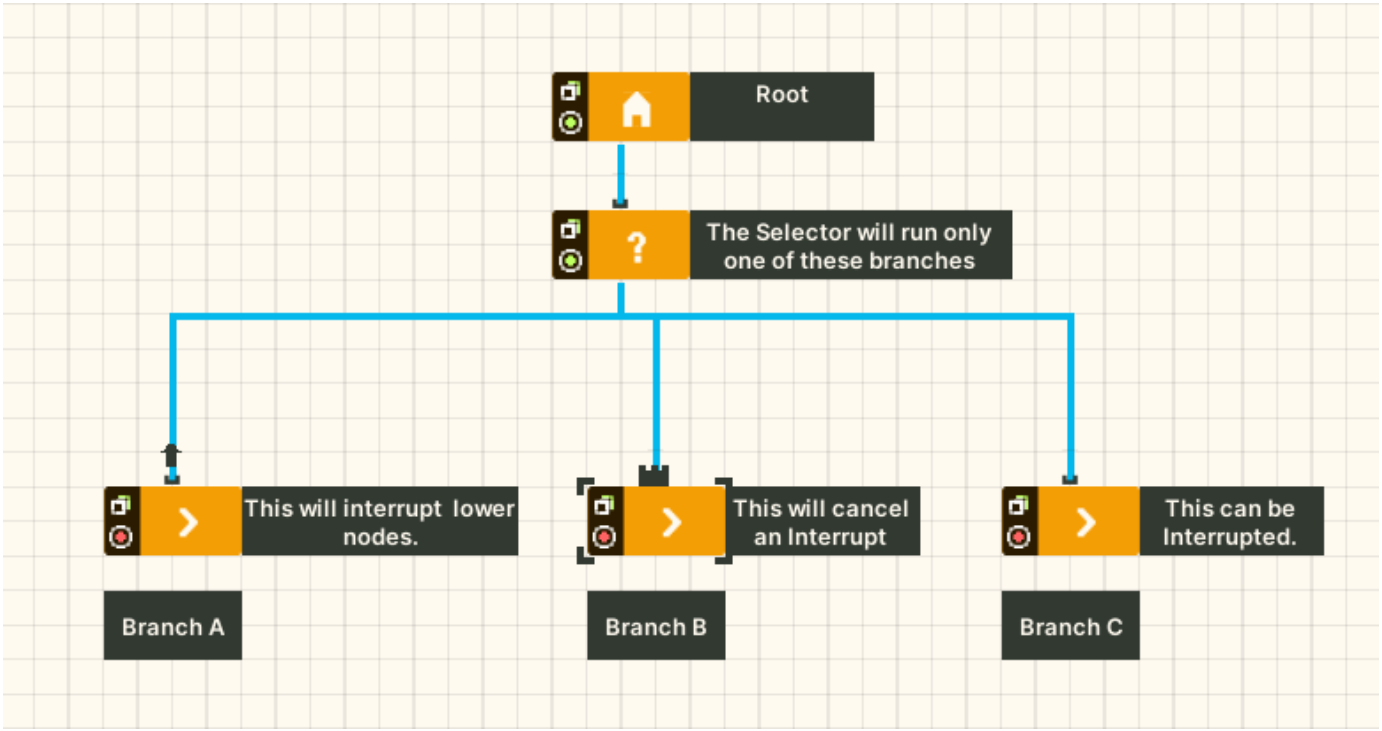
For example, if an AI is chasing the player but the player kills the AI in the process, and the node in charge of checking the health of the AI is no longer being checked, the AI will not know it is dead! To avoid this scenario, the higher priority branch should have an interrupt enabled so that the system is always checking the health of the AI.

It's also important to note that any branch can interrupt *itself* as well. For example, in the following scenario an AI is tasked with chasing the player but only if the player is inside the AI's territory. If the player is inside the territory, the AI will begin to chase the player blindly. The branch is no longer checking if the initial condition is true, so if the player steps outside the territory, the AI will not know and continue chasing! To avoid this scenario, the branch should enable a self interrupt to always check if the player is inside the territory.

An interrupt will only check the first child node of a Composite. If the child node is Conditional, the system will check for Success. If the child node is a Composite, it will go into the Composite and check if its first child is a Conditional, and so on. All other types of nodes are ignored.

Interrupts	
None	The node will not check for any interrupts.
This Node	The node will be able to interrupt its own branch. A downward arrow will appear above the node.
Lower Priority Nodes	The node will interrupt lower priority nodes. An upward arrow will appear above the node.
This And Lower Priority Nodes	The node will interrupt itself and lower priority nodes.
Terminate Immediately	If enabled and a lower priority node is interrupted, the node will terminate its job and allow the interrupt to continue.
Cancel Interrupt And Complete	If enabled and a lower priority node is interrupted, it will cancel the interrupt and continue executing. A black block will appear above the node.

Below is a contrived example. The nodes don't actually do anything, but it shows how interrupts work in theory.



Inspector

Property	
AI Type	Regular: the AI is affected by gravity and will be able to interact with the game world via raycasts. A BoxCollider2D will be required. No Collision Checks: this is the complete opposite of Regular. This is meant for AI that doesn't require complex interaction with the world. Moving Platform: if the AI is a moving platform, make sure it has this setting for proper function because moving platforms are executed before all other objects in the game world. Moving Platforms should not be rotated on their axis.

Collision And Gravity	Refer to player for these settings.
Damage	If enabled and if the AI has a Collider2D, it will deal damage to any object with a Health component that exists on the specified layer.
Create Units	If an AI is part of a group of units that operate under the same AI logic, use this to create the number of units (gameobjects) necessary. Every time you make a change to the FSM or BehaviorTree, recreate the units to ensure they all have the same code by pressing this button. The system will do its best to keep superficial transform settings the same.
Reset To First	For a FSM, if this is enabled the system will move to the first state on a global reset. This only occurs if the Reset State is empty. Otherwise the system moves to the first state automatically after the Reset State completes.

FSM

A finite state machine provides an intuitive approach to creating AI. States make it easy to reason about logic as long as the number of states remains small. Thus, using a FSM should be your first option when designing most basic AI.

All the nodes available to a Behavior Tree are also available to a FSM, except for Composite and Decorator nodes as those concepts are irrelevant here. Once you create a state, click the add button to open the node context menu and create the necessary nodes (depicted in orange).

There are three types of states to be aware of. First, you have the Normal States (depicted in blue). The first of these states will be the entry point into the state machine. When the state machine is running, only one of these states will be active at a time.

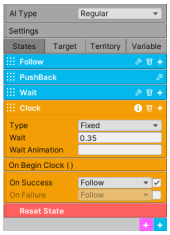
The state bar will have two important fields. The first field is for naming the state. Name the state a unique name for identification purposes. The second field lets you choose how to run the nodes. If **Parallel** is enabled, all the nodes will run at the same time. If **Sequence** is enabled, the nodes will run in sequence. That is, the system will not move to the next node until the current node either succeeds or fails. This sequence will always loop automatically. If **SequenceSucceed** is enabled, it works the same as Sequence except the system only moves to the next node if the current node succeeded. Create these states by clicking the blue add button at the bottom of the inspector.

Next you have the Always States (depicted in purple). These states will always run because sometimes it's necessary to have this type of functionality. It is not possible to jump states from an Always State. These states should contain content that are applicable to each state. You create these by clicking the purple add button at the bottom of the inspector.

And last is the Reset State (depicted in red). This state is called when the WorldManager performs a game reset. The Reset State, if used, occurs in one frame and does not check for collision. Use this state to reset the AI's position, health, and other important variables.

Since every node can potentially have one of three states (Success, Failure, Running), the system makes use of this to figure out when to jump to the next state. Thus, each node will come equipped with two options. On Success and On Failure. If either of those are enabled, and the condition is met, the system will jump to the specified state. However, keep in mind that not every node will return Success or Failure. Many Action Nodes only return Running and Failure, so they will not work to jump to a new state on Success, but some of them will return Success. When designing state jumps, don't lose sight of this fact.

Below is a simple FSM. The AI is tasked with following the player. If the player deals damage to the AI, the AI will be pushed back, and then it will go into the Wait State. There it will wait for the specified time. Once the clock timer is complete, the On Success option will trigger a state jump to the Follow State.



AI Attacks, Damage

There are four ways an AI can perform an attack on the player. Of course, if the following ways aren't what you are looking for, it is always possible to implement a custom solution.

First, in the Settings options enable Damage. The system will use the existing collider on the AI. Once the player comes into contact with the AI, it will be dealt damage.

Second, the AI can use a Firearm to shoot projectiles at the player. Somewhere in your logic, you will use an OnEvent node. This event should trigger the Shoot() method that belongs to the Firearm class.

Third, use the MeleeAttack node. Once this node is active, it will enable a separate collider to damage the player. The attack animation for the melee attack is setup and executed elsewhere. The MeleeAttack node is only in charge of dealing damage, enabling the collider, and setting the animation signals. You can use SpriteEngine to play the animation and control the size and position of the collider. Once the animation is done playing, the CompleteAttack() method of the MeleeAttack class **must** be called, or else the finite state machine will get stuck in its current state.

Fourth, use the **Damage** class. Simply add this component to any gameobject. Add a collider2D and set isTrigger to true. This will deal damage to any character that exists in the Damage Layer when contact is made. This is particular useful for static objects like spikes.

Nodes

The AI system comes with many built in nodes ready for creating AI. The following gives you a general understanding of the node's purpose. For more information on the node, hit the show information icon in the inspector.

Action	
Follow Target	The AI will follow the specified target.
Ignore Block Units	The AI will ignore or block other AI units along the pathfinding path.
Pathfinder	The AI will follow a path generated by the pathfinding algorithm. This works with gravity, making it ideal for platformers.
Pathfinder Basic	The AI will follow a path generated by the pathfinding basic algorithm. This ignores the effect of gravity.
Jump	Make the AI Jump.
Jump To	The AI will jump towards the specified target.
Push Back	If damaged, the AI will be pushed back (and flash).
Monitor Variable	Check if a blackboard variable has changed value. Typically used for float variables.
Monitor World Variable	Check if a world variable has changed value. This will also work with Health.
Apply Gravity	This will simulate gravity. Mostly needed by moving platforms.
Control Player	Control the player in the x-direction by moving it to a point. This will also block player input.
Idle	Sometimes it's useful to have an action where the AI just waits idly.
Melee Attack	Perform a melee attack.
Move	This will move the AI with the specified velocity.
Nullify Gravity	Stop the effect of gravity on the AI.
Rotate Around	Rotate the transform around the specified target.
Rotate Position	Rotate the transform's position around the center point at the specified radius.
Set Transform	Set the position, rotation, or scale of a transform instantly or by lerping.

Shake	This will shake the AI. Only shake an object if it's standing still.
Water Float	The AI will float on the waterline.
Face Target	The AI will point towards the specified target in the x-axis.
Get Nearest Target	Get the nearest target to the AI from a list and set the reference to this target.
Get Random Target	Get a random target from a list and set the reference to this target.
Nearest 2D Results	Get the nearest target to the AI from a list of physics results and set the reference to this target.
Target Changed	This will return success when the target's position has changed by the specified distance.
Add To List	Add the specified item to a list.
Animation Signal	Sets an animation signal true.
Chain	A chain created from gameobjects.
Clock	Wait the specified time.
Enable Behaviour	This will enable or disable the specified behaviour or renderer.
Enable Tree	Pause or remove the BehaviorTree from executing. Also specify the active state of the gameobject that belongs to the BehaviorTree.
On Event	Invoke and event.
PauseCollisions	Pause world collisions. If enabled, choose to move transform with Translate or not.
Remove From List	Remove the specified item from a list.
Set As Child	Set the specified transform as child of the AI or remove it as a child of the AI.
Set Value	Set the value of the blackboard data variable.
Set Value From blackboard	Set the value of the blackboard data variable with that of the blackboard from variable.
Sprite Color	This will modify a sprite's color by lerping to a new one.
World Effect	Call a world effect at the AI's position.
Damage Flash	Show damage flash.

Composite	
Parallel	Runs all child nodes at the same time. This will return Running as long as one of them is running. Once all child nodes complete, it will return Success if one child succeeded, else failure.
Parallel Success	Runs all child nodes at the same time until one of them returns Success. Returns Failure if all them fail.
Parallel Success All	Runs all child nodes at the same time until they all return Success. Will return failure if not all of them succeeded.
Random Selector	This will run one random child node.
Random Sequence	This will shuffle the list of nodes so that the execution order is always randomized.
Selector	This will run every node in the list until one returns Success.
Sequence	This will run every node in the list until one returns Failure.

Decorator	
Delay	Run the child node after a time delay.
Inverter	This will invert the output logic of the child node (except for Running). Interrupt logic will work with this decorator.
Repeater	This will execute the child node by the amount of times specified in the repeat value.

Running	This will always return Running.
Timer	This will run a child node for the specified time.
Until Fail	This will run the child node until it returns Failure.
Until Success	This will run the child node until it returns Success.

Conditional	
Find Target	Find the specified Target in relation to the AI or Territory.
List Logic	This will compare the size of the specified list to a value.
Variable Logic	This will compare a float variable to a float value.
World Float Logic	Compare a world float to a float value.
CircleCast	Implement a CircleCast using Physics2D. The results can be accessed by Nearest2DResults.
LineCast	Implement a LineCast using Physics2D. The results can be accessed by Nearest2DResults.
OverlapBox	Implement an OverlapBox using Physics2D. The results can be accessed by Nearest2DResults.
OverlapCircle	Implement an OverlapCircle using Physics2D. The results can be accessed by Nearest2DResults.
OverlapCollider	Implement an OverlapCollider using Physics2D. The results can be accessed by Nearest2DResults.
OverlapPoint	Implement an OverlapPoint using Physics2D. The results can be accessed by Nearest2DResults.
RayCast	Implement a RayCast using Physics2D. The results can be accessed by Nearest2DResults.
Layer Result	Rays casted using Single Hit can compare if the resulting object belongs to the specified layer.
Touching Layers	Check if the specified collider is touching any other collider in the specified layer.
Collision Status	Check what a character/AI is interacting with.
Return Failure	This will always return Failure.
Input Get	Get Input KeyDown or MouseDown.
Positional	Check the AI's position in relation to a target.
Field Of View	Returns Success if the specified target is inside the field of view.
Has Passengers	Does this moving platform have passengers?

Create Nodes

The most common nodes to create will be Conditional and Action nodes. Once the script is created, place it inside the AI/BehaviorTree/Nodes folder and it will become available for use in the AI system. Follow the template below to code your own functionality. The first example is of the Move Action node, which simply adds velocity to the AI.

```
using UnityEngine;

namespace TwoBitMachines.FlareEngine.AI// Include this namespace
{
    public class Move : Action // Specify the type of node - Action/Conditional/Composite/De
    {
            [SerializeField] public Vector2 velocity;
```

```
            // Use this method to implement the behavior
            public override NodeState RunNodeLogic (Root root)
            {
                    if (nodeSetup == NodeSetup.NeedToInitialize)
                    {
                    //       Any member fields that need to be reset/initialized go here.
                    }

                    root.velocity += velocity;// Root is basically the AI
                    return NodeState.Running; // Since this is an Action node, return Runnin
                                              // If the behavior completes, return Success o
            }

            // Use this to reset important variables
            public override void OnReset ( )
            {

            }

// Root contains a few variables to be aware of

// root.velocity  -- the velocity of the AI, read and write to it
// root.direction -- the direction of the AI on the x-axis, read and write to it
// root.position  -- the position of the AI in the game world, read only

// root.hasJumped -- set true if you have added a jumping force to the AI's root.velocity.y
// root.onSurface -- set true if you are creating a hard surface for the AI to stand on

// root.signals.Set ("relevantSignalName") -- set an animation signal if necessary
// root.world     -- reference to WorldCollision, read settings like onGround, onMovingPlatf
// root.gravity   -- reference to Gravity, if jumping use root.gravity.SetJump (velocity.y);
//                   see the Jump node for more information
// root.movingPlatform   -- reference to MovingPlatform if the AI is of this type, use it to
//                          hasPassengers or passengerCount
    }

  }

using UnityEngine;

namespace TwoBitMachines.FlareEngine.AI
{
     // This is a Conditional node. This will simply check if the AI is on the ground.
     // This class doesn't actually exist, but it can be created using this code
     // to get this functionality.
     public class AIOnGround : Conditional
     {
              public override NodeState RunNodeLogic (Root root)
              {
                      return root.world.onGround ? NodeState.Success : NodeState.Failure;
              }
     }
 }

using UnityEngine;

namespace TwoBitMachines.FlareEngine.AI
{
```

```
    // This is a Decorator, and it will run its child node until it fails.
    public class UntilFail : Decorator
    {
        public override NodeState RunNodeLogic (Root root)
        {
            NodeState nodeState = children[0].RunChild (root);
            return nodeState == NodeState.Failure ? NodeState.Success : NodeState.Runn
        }
    }
}
```

Pathfinding

Pathfinding and **PathfindingBasic** are a-star algorithms that come included with the AI system. They allow AI agents to follow generated paths towards a target. The main difference between both classes is that Pathfinding takes gravity into consideration, so it's ideal for AI agents that need to jump from platform to platform. It has many other advanced features, too. The AI agent can be made to climb ladders, walls, ceilings, and can walk on bridges. PathfindingBasic ignores gravity and is meant for general use in a 2D world. Both systems can be found in the Territory tab for both AITree and AIFSM.

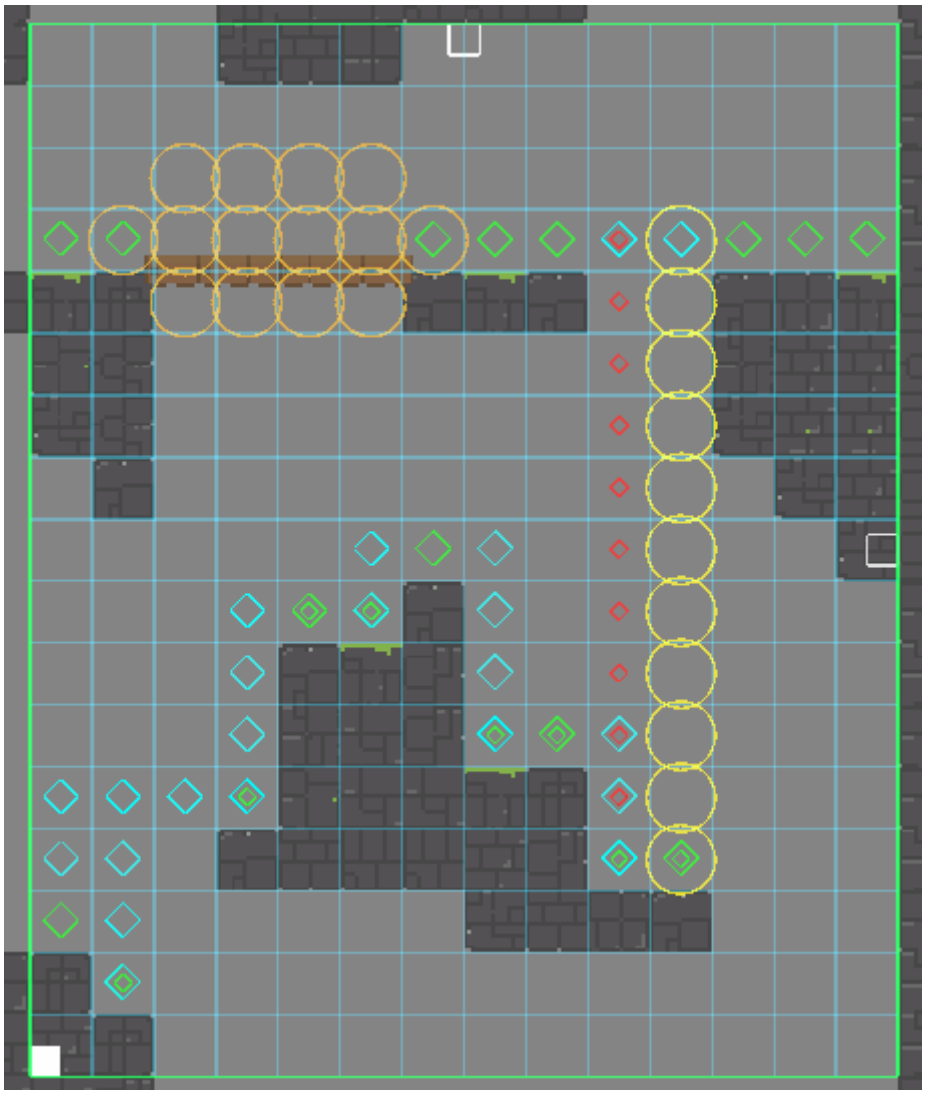
Once a pathfinding territory is created, a green rectangular gizmo will appear in the Scene. Drag and resize this rectangle over gameobjects that contain Collider2Ds and the correct layer (usually World). This will become the area where paths are generated.

Pathfinding	
Data Name	The name of the pathfinding territory.
Layer World	Raycasts will be used against this layer to create the pathfinding terrain. If this is not set, nothing will work.
Max Jump Height	The maximum height the AI agent can jump to a platform in the y-direction.
Max Jump Distance	The maximum distance the AI agent can jump to a platform in the x-direction.
Cell Size	The size of each cell in the grid.
Show Paths	If enabled, the relevant path nodes and grid will be visible.
Add Ladders, Walls, Ceilings, Bridge	When this is enabled, go to the grid and click on a cell where you wish to add this feature. To remove, simply click on the cell again. Walls and Ladders must be created with vertical lines. Ceilings and Bridge must be created with horizontal lines. Bridges need extra circles to ensure the AI agent doesn't sag below due to the bridge. When complete, disable this toggle, and then press the Create Paths button to include the new changes.
Create Paths	When pressed, this will create the necessary path nodes. Every time you make a change to these settings, you must repress this button.

When the create button is pressed, the relevant path nodes will appear. The green diamonds represent ground. This is where the AI can walk. The blue diamonds represent jumping connections. If two platforms are not connected by blue diamonds, the AI cannot jump between the platforms. The red small diamonds are always vertical and they represent paths where the AI can fall from. These paths are too high for the AI to jump to, but they are still available for the AI to fall from. The yellow circles represent a ladder (not depicted). The orange circles represent a bridge. The other features will include their own distinct colors.

Warning

If using a Tilemap with a Composite Collider 2D, creating the paths will not work since these colliders can't detect raycasts from the inside. For a quick fix, change the Geometry Type to Polygons, press the create paths buttons, then revert back to the original Geometry Type if desired.



Here we can see an AI agent (red square) follow a path to reach the player. The red circles represent the path the AI agent is following.

Once a pathfinding class is created, leave it alone on its own gameobject and don't add anymore nodes or states. Its only job is to generate paths. This will be useful in case there are multiple AI agents referencing the same pathfinding class. To create an AI agent that will follow a generated path, create another gameobject and add an AI system. Create two targets. One target will be the end point of the path, usually the player. The other target is **Target Pathfinding**. This target will hold a reference to the pathfinding class, specify follow speeds, and will requests for paths to be generated.

Target Pathfinding	
Data Name	The name of the pathfinder.
Map	The reference to the pathfinding class.
Speeds	The relevant speeds the AI agent will use along the path.
Ignore Units	If disabled, the AI agents will *try* to block each other along the same path in certain situations. This is to prevent AI agents from overlapping. Other ways to prevent overlapping is to give each AI agent a slightly different speed.

Pause After Jump	Every time the AI agent performs a jump, it will pause after by the specified amount before continuing to move along the path. This allows the agent's movement to feel a bit more natural.

The last step is to create the **Pathfinder** node. This will be created inside a state if using AIFSM or as a tree node if using AITree.

Pathfinder	
Pathfinding	The reference to Target Pathfinding.
Target	The reference to the target the AI agent is trying to follow.
Reset Distance	When the target changes its position by this amount, the system will generate a new path for the AI agent to follow.
Success On	If enabled, this node will return Success if the condition has succeeded.

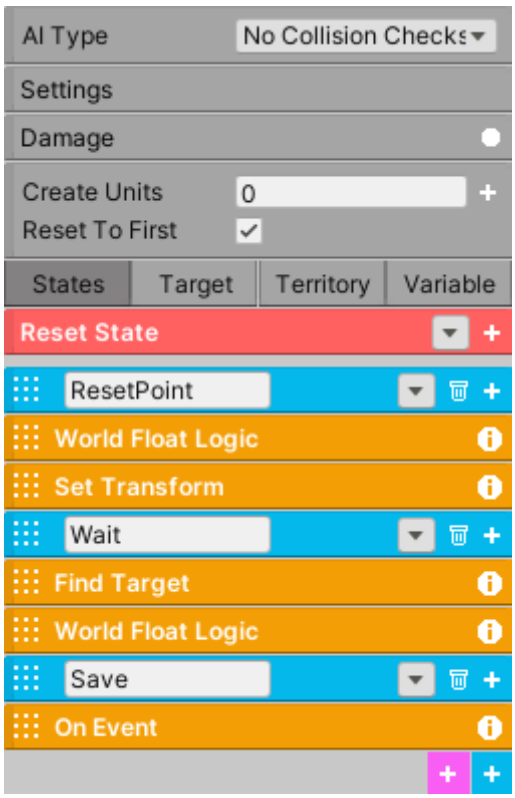
Note

The steps to setup **PathfindingBasic** will be similar but with less settings due to its simplistic nature.

Check-Point

On a game reset, the system will reset the player's position at the desired location, saving player progress.

The check-point exists as a prefab and requires some setup. The check-point system was actually created using the AI system. We will break-down how it works internally and hopefully this can serve as a good case study and give you an idea on how to create your own systems.



The system uses a AIFSM, for keeping track of the player, and a World Float, for saving the state of the check-point system.

Before explaining the states, first let's see how the system is designed. Each level can have as many check-points as desired. For this particular scenario there are three check-points. Each check-point will be placed in the game world and each one will have a corresponding arbitrary index. So 1,2, and 3.

Once the player goes inside a check-point area, the FSM will save its index in the World Float (which will have Save enabled). If the current value of the World Float is less than the check-point index, the value will be updated. This means the check-point system works in terms of priority. If the player goes back inside check-point 1 but has already been to check-point 2, the system will not revert back to check-point 1. And on a game reset, the AIFSM will move the player to the desired location of the check-point that matches the saved value with its index.

How are the check-points aware of each other?

World Float is the glue that holds everything together. Each check-point comes with a World Float, and each World Float must have exactly the same name as all the others. If each World Float has the same name, it means they are saving to the same variable in memory, so they all reference the same data. And equally important, each time a World Float is modified it will send a signal to each World Float that shares the same name and it will update their values to match. For this broadcast signal to work, the Broadcast Value must be enabled for each World Float. This will ensure all the World Floats are working with the same value.

How does the Check-Point FSM work?

The FSM has three states. The first state, ResetPoint, will check if the World Float matches its index. If it does, the state will reset the player's position using Set Transform. Since this is the first state, the AI system will always check this state first on scene start. On a game reset, enable Reset To First in settings, which means the AI system will move back to the first state on a game reset and reset the player's position.

Regardless of what happens in ResetPoint state, the system will move to the wait state. Here, the system will wait until the player enters its territory using Find Target. If the player does enter its territory, it will compare the World Float current value with its index. If the World Float current value is less than the current index, the system will move into the Save state and save the index into the World Float and then revert back to the Wait state.

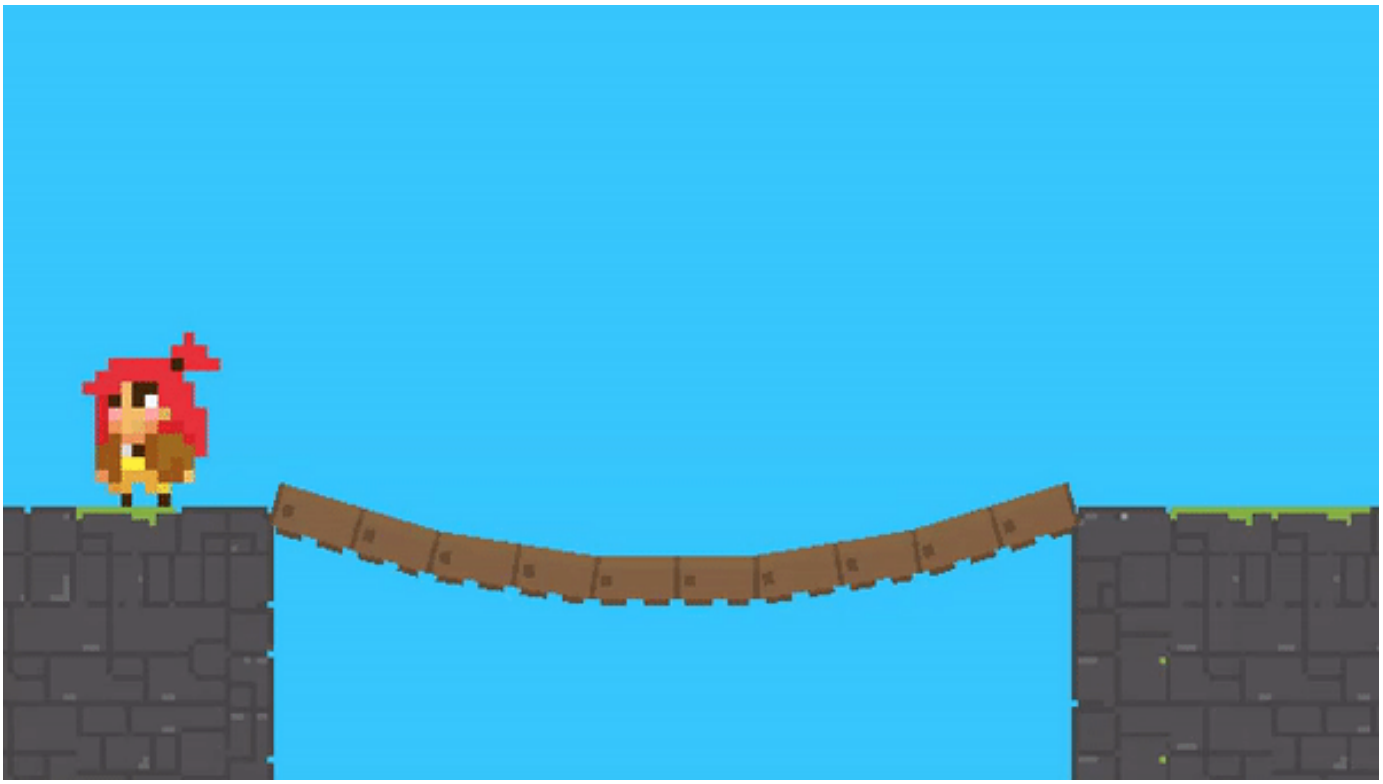
It's important to note the Wait state is set to Sequence Succeeded, which means the state will not check World Float Logic until Find Target has returned success.

Important

The parts you must modify: in Set Transform, input the reset position. In each World Float Logic and On Event, place the index of the check-point. You will also need to move each territory (green rectangle) manually in the scene for each check-point.

Bridge

A bridge creates dynamic movement. Characters can walk and jump on them. Place this component on an empty gameobject and scene gizmos will appear. The gameobject's position is the start position of the bridge, and the red gizmo will modify the end position.



Tip

Characters, by default, are enabled to interact with bridges. If this property is not desired, disable it in the character's collision settings to save unnecessary collision checks.

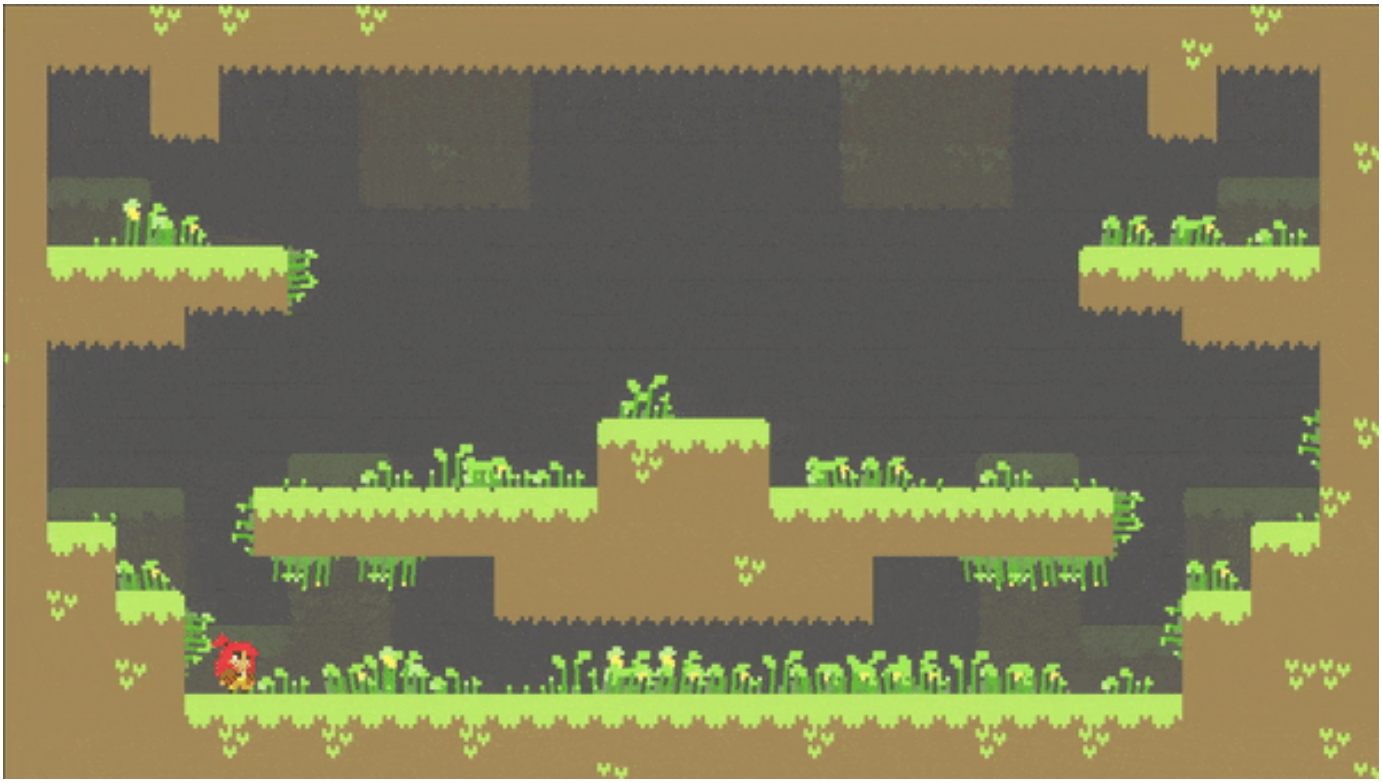
Property	
Planks	The number of planks in the bridge.
Gravity	The force of gravity acting on the bridge.
Bounce	The force exerted on the bridge when interacting with characters.
Stiffness	The larger the number, the less sag the bridge will have. For performance, keep this value below 30.
Plank	The system will create a gameobject for a plank with the corresponding sprite. This will be used as a template to instantiate the remaining planks. You can change the transform's scale to achieve the desired plank width. The offset will shift each plank visually.
Area	The system will check for plank collisions once the character is inside the bridge area. The area width is set automatically, but the height must be specified. The offset will offset the area in the y direction.
Create	Once all the settings are chosen, press this button to create the bridge. Anytime you change a position or a setting, recreate the bridge to enact the changes.
View	If enabled, the bridge gizmos will be visible.

Important

The start of the bridge corresponds to the transform's position. Make sure the transform's handle position is set to Pivot (and not to Center) for proper placement. A scene handle tool, a red circle, is used to specify the end of the bridge. The distance between the start and end points determines the length of the bridge.

Foliage

Decorate an environment with foliage to make it come alive. The foliage can sway with the wind and respond to all characters. Add this component to an empty gameobject.



Property	
Jiggle	The motion effect produced when interacting with characters. Smaller values produce softer motions.
Damping	How quickly the jiggle effect dissipates.
Uniformity	The tendency for foliage to sway in the same direction if the foliage has the same y position.
Wind Strength	The force of the wind swaying the foliage.
Wind Frequency	How quickly the wind changes direction.
Create Texture	Press this button to add a new Texture2D. This is the foliage. Each Texture2D must have the same size as the specified Vector2 field.

Warning

The system groups all the Texture2D images of the foliage into an array. Thus, every Texture2D must be of the same size and share the same settings for this process to work correctly. As a reminder, this component is working with Texture2D and not Sprites.

Texture2D	
Texture2D	The current Texture2D image of the foliage. The delete button will remove this Texture2D and all of its instances from the scene.
Orientation	This determines what vertices to sway. If Bottom is enabled, place foliage on ground. If Top is enabled, place foliage on a ceiling. If Left or Right are enabled, place foliage on walls.

Depth	Specify the rendering order of the Texture2D images relative to each other. As of now, there is no way specify a sorting layer. Characters are either in front or in back of the foliage – never in between.
Interaction	Choose how active the foliage is with character interactions. Maybe some foliage are dense and don't need to sway as much as others. A value of zero will disable all interactions with characters.

Paint Brushes	Place foliage in the scene with brushes.
Single Brush	Place a single foliage image.
Random Brush	Choose as many foliage images as desired and drag the brush in the scene. The density value specifies how many images the brush can place per position.
Eraser	Use this brush to erase foliage images. Please note, some foliage instances will still be visible even after erased. Sometimes it takes Unity time to remove them completely.
Instances	Every Foliage component can only have a maximum of 1023 images in the scene.

Tip

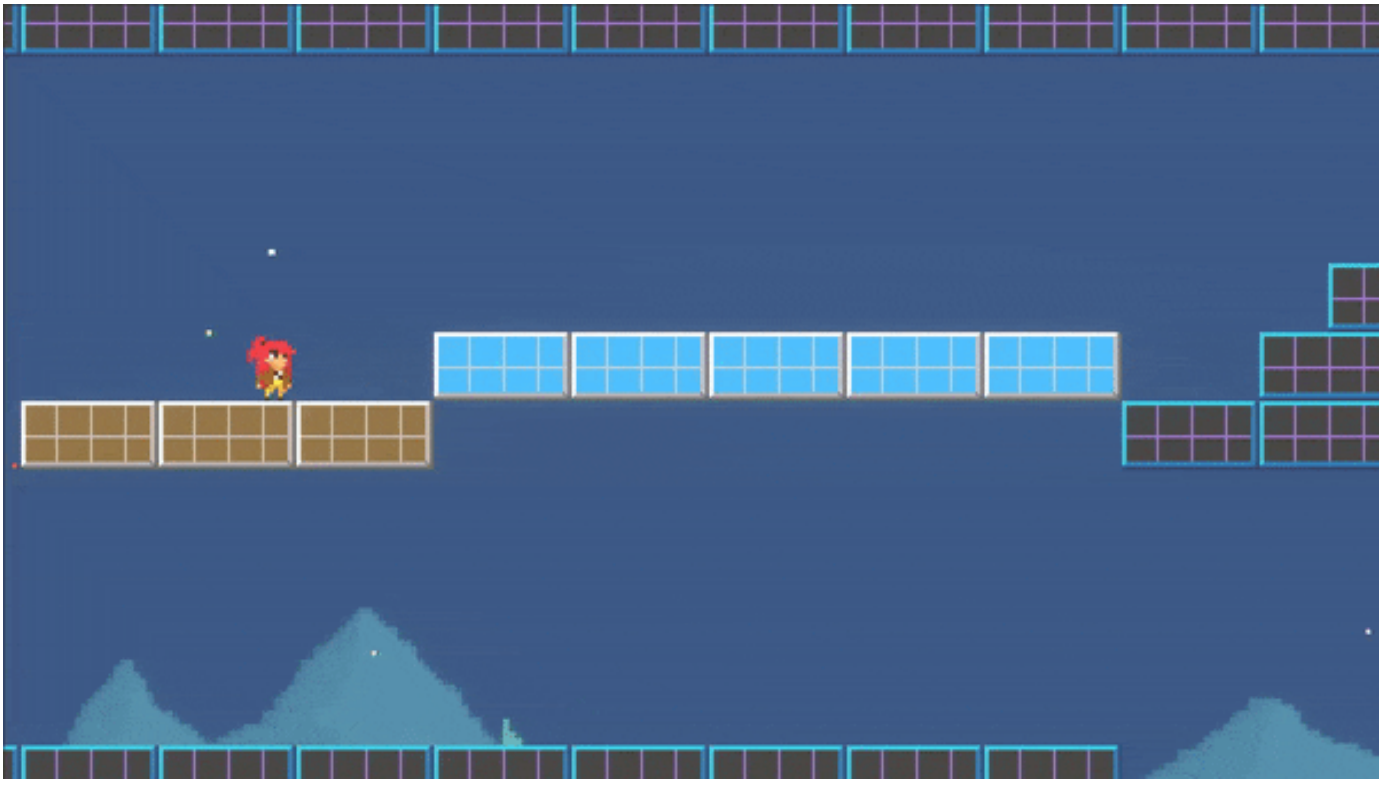
If the brush tool is active, right click in the scene or repress the current brush button to deactivate it.

Note

The foliage system was designed with performance in mind. All foliage instances exist in code only (they're not gameobjects), and the character interactions are handled by Unity's Job System.

Friction

Change the friction properties of the ground. Add this component to any gameobject requiring this feature and set its Tag to Friction.



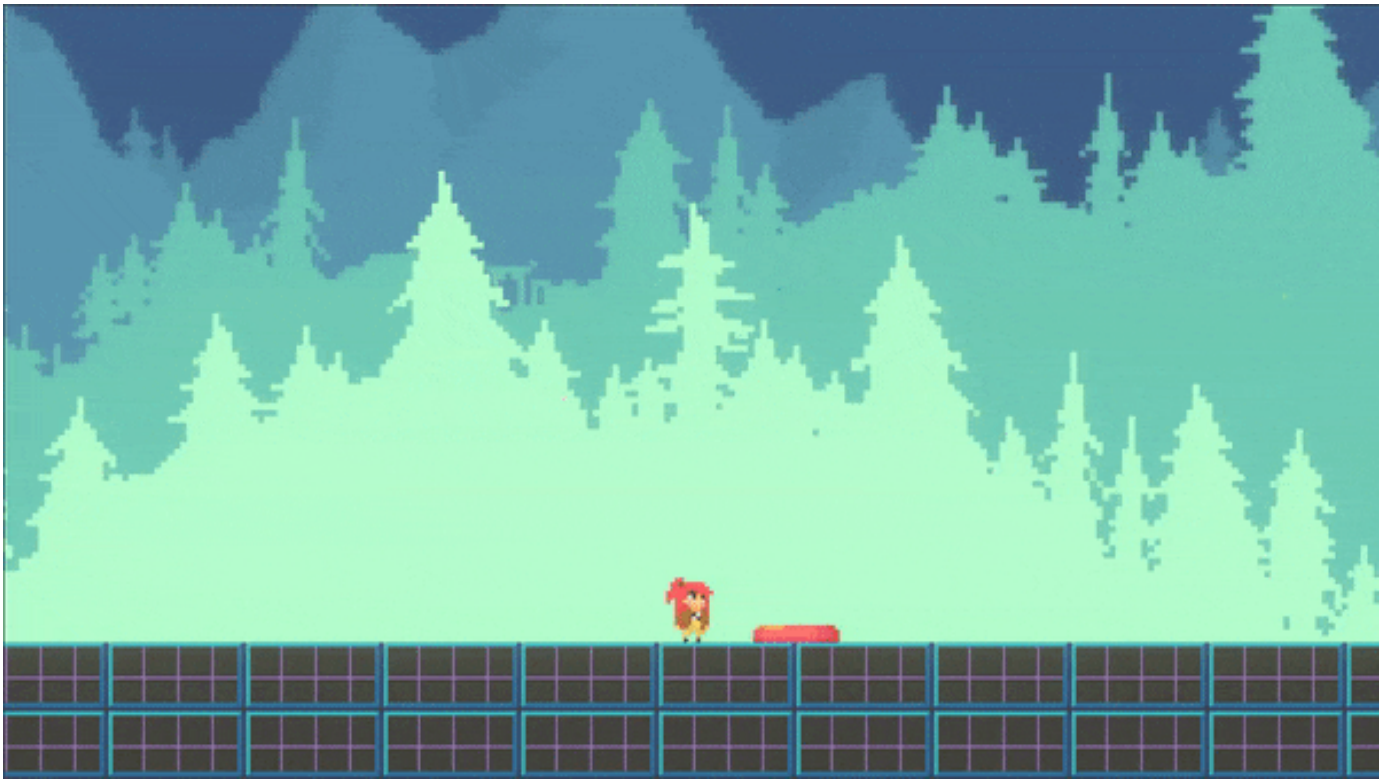
Note

The player's Ground ability must be enabled.

Property	
Type	Friction, Slide, Auto.
Friction	The amount of friction on the ground.
Slide Speed	The slide speed on the ground.
Auto Speed	The speed and direction in which the player will automatically start moving while on the ground.

High Jump

Launch a character into the air. This has two modes, Trampoline and Wind. Place this component on an empty gameobject. A blue rectangle gizmo will appear in the scene to modify its bounds for character detection. Typically, a trampoline will have a smaller collider area, and Wind will have a much taller collider area.



Note

High Jump must also be enabled in the character settings.

Property	
Type	Launch the character with Trampoline, or gradually push the character upward with Wind. Wind typically has small values below 1. Trampoline has big values above 10.
Force	The amount of force acting on the character.

On Trampoline	The Unity Event invoked when trampoline is triggered. Call a World Effect with the dynamic Activate method.

Ladder

The humble ladder is used for climbing. Add this component to an empty gameobject.

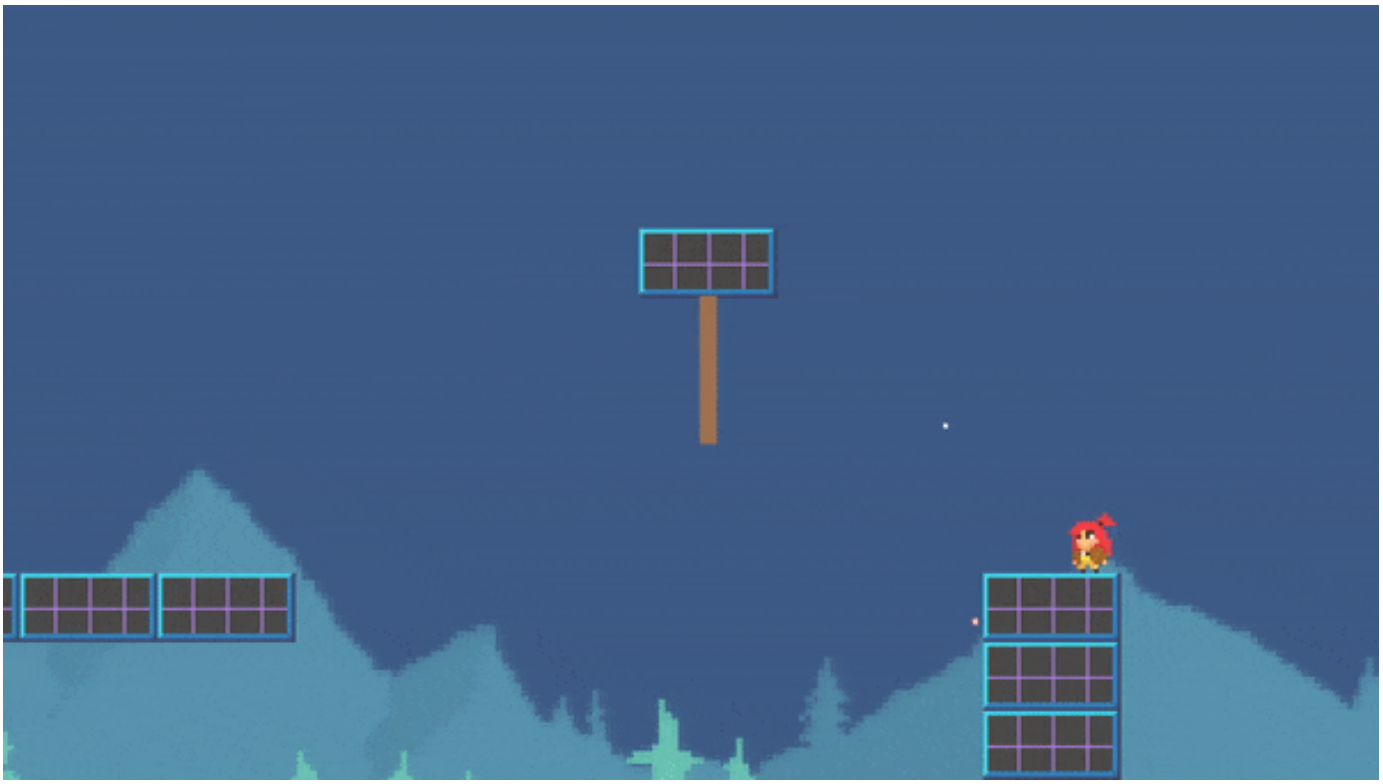
Note

The player's Ladder ability must be enabled to interact with ladders.

Property	
Size	The width and height of the ladder.

Rope

The player can use ropes to swing or for simple idle interactions. Add this component to an empty gameobject.



Note

The player’s Rope ability must be enabled to interact with ropes.

Property	
Type	If Swing is enabled, the player will swing on the rope. If Idle is enabled, the player will pass through the rope, causing it to move.
Rope End Radius	If Swing is enabled, once player and end tether are within this radius, the player will latch onto the rope automatically.

Type Idle	
Rope Radius	The system will check for rope collisions if the player is inside this radius. The center of this radius is set automatically.
Tether Radius	It is the radius of each tether used to detect the player.
Force	It is the movement force applied to a tether upon interaction.

Property	
Tethers	The number of tethers in the rope.
Gravity	The force of gravity acting on the rope.
Stiffness	The larger the number, the less sag the rope will have. For performance, keep this value below 30.
Double Anchor	Both the start and end of the rope are anchored.
Rope Sprite	The system will create a gameobject for a tether with the corresponding sprite. This will be used as a template to instantiate the remaining tethers. The second field corresponds to the size of the tether.

Rope End (Optional)	Every time the rope is created, it destroys and recreates all the tethers. Sometimes the end tether contains components like Health. To prevent having to add these components every time the rope is recreated, specify the end tether gameobject to prevent it from being destroyed.
Create	Once all the settings are chosen, press this button to create the rope. Anytime you change the rope's position or a setting, recreate the rope to enact the changes.
View	If enabled, the rope gizmos will be visible.

Important

The start of the rope corresponds to the transform's position. Make sure the transform's handle position is set to Pivot (and not to Center) for proper placement. A scene handle tool, a red circle, is used to specify the end of the rope. The distance between the start and end points determines the length of the rope.

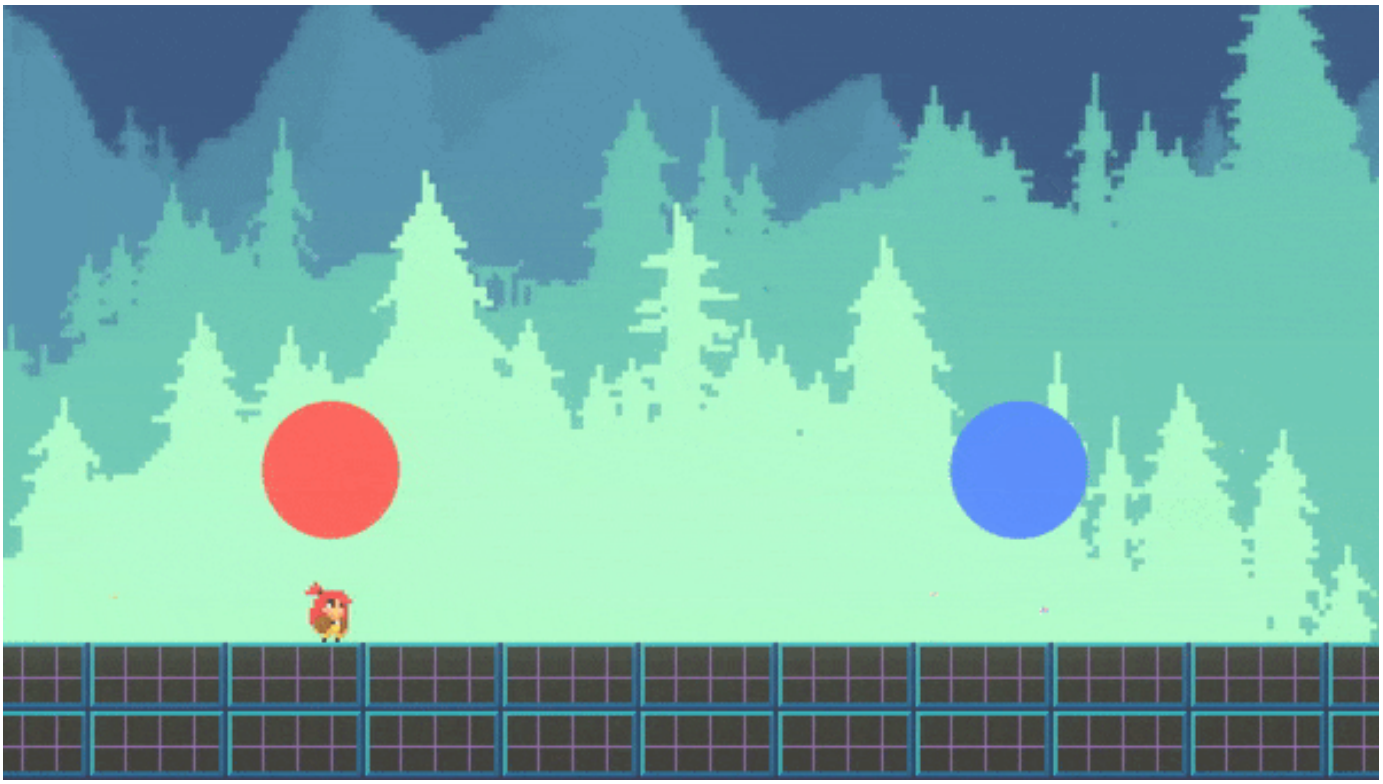
Method	
ApplyImpactAtEnd (float directionX, float impact)	This will apply an impact force in the x direction to the end of the rope. This is automatically used by the player for swinging.
ApplyImpact (float value, Vector2 direction)	Each tether contains the component Tether. This class contains this method. Call it to apply a force to a tether in the specified direction. Ignore the value parameter and instead set the impact force in the inspector field of the Tether class.
UnlatchEndAnchor ( )	If double anchor is set true, you can set it false by calling this method. The end anchor will become free, letting the rope fall down.

Tip

It's possible to add a Health and Collider component to each tether for further interaction. This can be useful if the rope needs to collide with Projectiles. The Health component is equipped to call the ApplyImpact() and UnlatchEndAnchor() methods through Unity Events.

Teleport

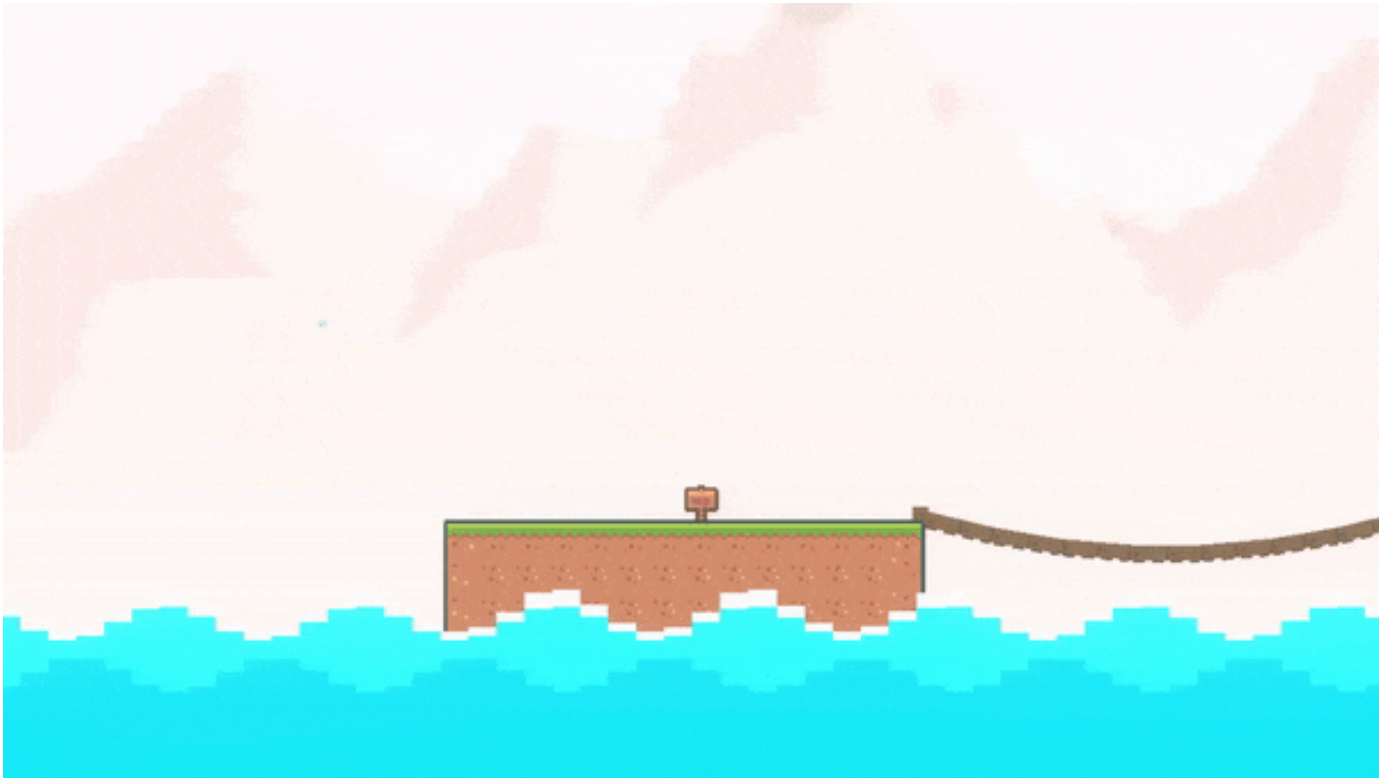
Any gameobject that enters the collider2D on any gameobject with the Teleport component will be teleported to the specified destination. It is perfectly possible to create bidirectional teleports that work together.



Property	
Destination	The transform where the gameobject will be teleported to.
LayerMask	Only gameobjects on this layer mask will be teleported.
On Teleport	The event invoked when a gameobject is teleported.

Water

Water is a dynamic area where the player can float and swim. Add this component to an empty gameobject.



Note

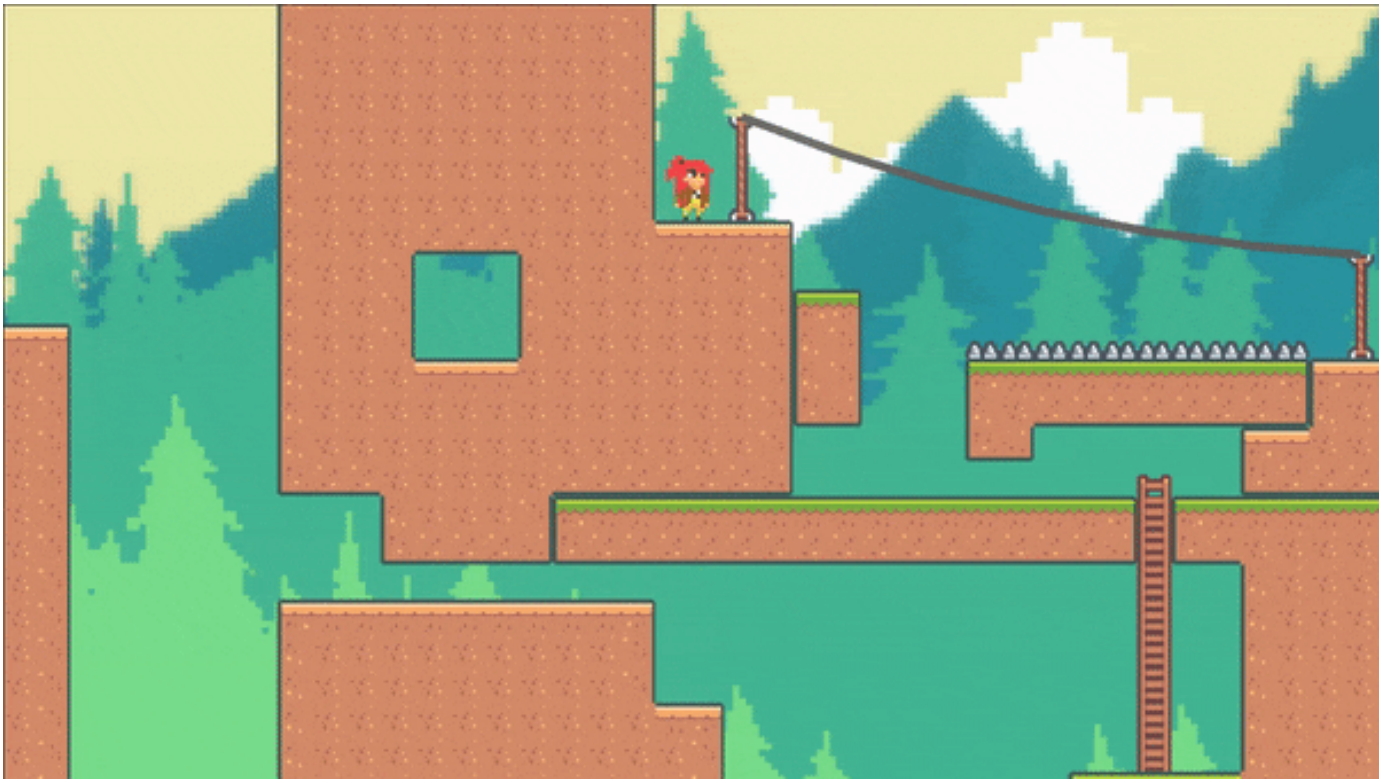
The player’s Swim ability must be enabled to interact with water.

Property	
Shape	If Square is enabled, the system renders the water using square blocks. No textures or sprites are required. However, the sorting layer will always be set to default (a Unity limitation). If Round is enabled, the system renders the water using a Mesh Renderer, creating a more curved wave. A Texture2D and Material are required.
Type	If Float is enabled, the player will stay above the water line. If Swim is enabled, the player can swim inside the water.
Segments	The number of elements that create the water. The higher the number, the less blocky the water will look.
Texture2D	If Shape mode is Round, provide the Texture2D that will be used to render the water.
Material	If Shape mode is Round, provide the Material that will be used by the Mesh Renderer.
Amplitude	The maximum height of the wave.
Frequency	Dictates the number of waves in the water.
Speed	How quickly a wave moves across the water.
Spring	The force exerted on the water when interacting with the player.
Damping	How quickly the spring force dissipates.
Turbulence	This adds random noise into the water, creating a chaotic effect.
Random Current	This will change the direction of the speed at intervals specified by this value. This value is randomized slightly to add unpredictability.
Create	Once all the settings are chosen, press this button to create the body of water. Anytime you change the water’s position or a setting, recreate the water to enact the changes.

Body	For Square mode.
Top	The color of the water line.
Thickness	The thickness of the water line.
Taper	The wave's water line will be thicker at its crest, and thinner at its trough.
Middle	The color at the middle of the water.
Bottom	The color at the bottom of the water.
Phase	The bottom of the water has wave like motion as well. Specify the phase of this wave.
Offset	Offset the position of the bottom wave.
Speed	How quickly the bottom wave moves across the water.

Zipline

Allow the player to zipline. This component works directly with Unity's Line Renderer. Modify the necessary settings there, mainly the width and color of the line. The player ability, **Ziplining** must be enabled. Place this component on an empty gameobject and scene gizmos will appear. The gameobject's position will be the start position, and a red gizmo will modify the end position.



Note

Don't forget to add a material to the Line Renderer (Default-Line).

Property	
Lines	The number of lines in the zipline.
Gravity	The force of gravity acting on the zipline.
Up Friction	The friction exerted on the player if moving up the zipline. To ignore, leave as zero.

Bounce	The force exerted on the zipline when interacting with the player.
Stiffness	The larger the number, the less sag the zipline will have. For performance, keep this value below 30.
Line Renderer	The reference to Line Renderer.
Area	The system will check for ziplines once the player is inside this area. The area width is set automatically, but the height must be specified. The offset will offset the area in the y direction.
Create	Once all the settings are chosen, press this button to create the zipline. Anytime you change the position or a setting, recreate the zipline to enact the changes.
View	If enabled, the zipline gizmos will be visible.

Important

The start of the zipline corresponds to the transform's position. Make sure the transform's handle position is set to Pivot (and not to Center) for proper placement. A scene handle tool, a red circle, is used to specify the end of the zipline. The distance between the start and end points determines the length of the zipline.

Audio Manager

The **AudioManager** conveniently stores all the music and sound effects for the current scene. It requires two audio sources that must be added manually. One will be dedicated to playing music, and the other will be dedicated to playing sound effects.

The Audio Manager component and the audio sources can be added to the World Manager's gameobject. In the inspector, go to Audio Category, specify a category name, and click the add button. This will create an audio category, which is meant to organize sounds into groups for convenience. Each category has an add button to create

a sound item. In each sound item you can specify the audio clip, volume, type, and can even play the audio clip in the inspector.

To play an audio sound call the provided methods. Since the engine contains plenty of Unity Events, it's possible to play audio from almost any system. And as such, **AudioManagerSO** is a scriptable object that can be created to hold a reference to Audio Manager, which means you can drag and drop Audio Manager SO into any Unity Event instead. This will allow you to play audio in the same way while using the flexibility of a scriptable object. To create one right click Create/FlareEngine/AudioManagerSO.

Property	
Music	The reference to the audio source that will play music.
Music Master Volume	This will scale the sound level of each music item. A value of zero means there's no music sound.
SFX	The reference to the audio source that will play sound effects.
SFX Master Volume	This will scale the sound level of each sfx item. A value of zero means there's no sfx sounds.
Fade In Time	The amount of time to fade in music.
Fade Out Time	The amount of time to fade out music.
Audio Manager SO	The scriptable object reference for playing audio.
Audio Category	Create an audio category with the specified name.

Method	
PlayAudio (string audioName)	Play this audio. This can be a music or sfx item.
FadeInMusic (string musicName)	This will gradually fade in the music sound instead of playing it at its maximum level immediately.
FadeOutMusic ( )	This will gradually fade out the current music playing.
FadeToNewMusic (string musicName)	This will fade out the current music and fade in the new music.
MusicVolume(float value)	Change the current sound level for music. Must be a value between 0 and 1f.
MasterMusicVolume (float value)	Change the master volume for music. Must be a value between 0 and 1f. This value will be remembered by PlayerPrefs.
SFXVolume(float value)	Change the current sound level for sfx. Must be a value between 0 and 1f.
MasterSFXVolume (float value)	Change the master volume for sfx. Must be a value between 0 and 1f. This value will be remembered by PlayerPrefs.
StopMusic()	Stop playing music.
StopSFX()	Stop playing sfx.
PauseAllAudio()	Pauses all audio sounds.
UnpauseAllAudio()	Unpauses all audio sounds.
StopAllAudio()	Stops all audio sounds.

Camera Engine

The camera will follow the player with a smoothing speed.

Property	
Target	The transform the camera will follow.
Speed	The follow speed. If smoothing is 1, the follow speed will be instant in that direction.

Smooth X	The smoothing effect in the x-direction.
Smooth Y	The smoothing effect in the y-direction.
Clamp	If enabled, these bounds will clamp the world position of an orthographic camera. Modify the blue gizmo in the scene to specify this area.

Rooms

Section your game world into rooms. Once the player enters a room, the camera will start to transition into it. The room will then clamp the camera's position to prevent it from escaping the room's boundaries.

Add a room in the inspector and modify the rectangular gizmo in the scene to change the room's size and position. Clicking the upper right button will change the room's size to match that camera's size for convenience.

It is also possible to modify the detection area of the room. These gizmos are found at the center of the room. Once you move them a red line will appear. This red line will signify the position the player must cross in order to trigger a room transition. Changing the detection area is optional. If left alone, the detection area will default to the room size. Regardless if the detection area is modified or not, the camera will always be clamped by the room walls.

It is also possible to overlap two rooms that share a common entrance area.

Property	
Transition Tween	The tween that will be used during a room transition.
Transition Time	The duration of the room transition.
On Enter	The Unity Event invoked when the player enters a room.
On Transition Complete	The Unity Event invoked when the room transition completes.
On Exit	The Unity Event invoked when the player exits a room.

Parallax

Implement a parallax effect with infinite scrolling. Create the necessary parallax layers by clicking the add button. Each layer will correspond to a gameobject which contains a Sprite Renderer (set the layer order manually).

In the inspector, set the reference and the parallax rate, which is the rate at which the layer will move with the camera. A value of 1 means the layer moves exactly with the camera. This is typically the value for the farthest layer, the background. A value of zero means the layer is standing completely still. This is typically the value of the nearest layer, the foreground. The parallax rate for the y values should typically be smaller than the x values.

These layers can be organized outside the camera gameobject. That is, they are not required to be children of the camera.

Shake

Shake the camera!

Method	
Shake (float strength, float duration, float speed)	Shake the camera with the specific values.

Shake	A convenient method to shake the camera with predefined values of strength = 0.25, duration = 1f, speed =2f;
SmallShake	A convenient method to shake the camera with predefined values of strength = 0.05, duration = 0.5f, speed =2f;

It is also possible to call these methods statically. Call their alternate methods CameraEngine.ShakeNow(params), CameraEngine.ShakeNow(), and CameraEngine.SmallShake(). You will need to include the TwoBitMachines.FlareEngine namespace.

Dialogue

This is an interactive dialogue system for the player and NPCs. Create conversations, add message effects, call Unity Events, and even include simple animations for complex interactions. All this is accomplished within the inspector window.

The setup requires at least two components: **Dialogue**, where the dialogue between the characters is created and stored, and **DialogueUI**, which controls the UI elements that will display the dialogue. The Dialogue component will typically exist on an NPC, an empty gameobject, or wherever it is needed, and the DialogueUI should exist on a canvas element, to properly reference all the necessary UI elements. If the UI and message need special effects, the **LetsWiggle** component, a tweening library, can move and scale UI elements, and the **TextMeshProEffects** can add a typewriting effect, among other effects, to the message.

Note

When dialogue is started, player input will be automatically blocked.

Property	
DialogueUI	The reference to DialogueUI.
Exit Button	If this button exists, the user will be able to escape any dialogue.
Skip Button	If this button exists, the user will be able to skip to the next message in the conversation.
Position Icon	If enabled, the messenger icons will be positioned on the left or right of the message box, depending on the relative position of the messengers.
Is Random	If enabled, this will display random conversations. If this is desired, enable the Is Random flag that also exists in each conversation for any conversation that should be treated as random.

Method	
BeginConversation( )	Call this method to begin the conversation. This will also set the gameobject active true.
EndConversation()	Call this method to end the conversation.

Create any messenger that is relevant to the dialogue.

Property	
Messenger Bar	Choose the messenger's sprite icon and display name. From here, delete this messenger if necessary, and toggle the box open for more fields.
Transform	If Position Icon is enabled, specify the transform of the messenger to compare its position relative to the other messengers.

Background	This is optional. Choose a personal background image for this messenger. If left null, the default background will be used instead.
Create Animation	If applied, the messenger's icon will be replaced by an animation. Name the animation and create it. Create as many as necessary. Once created, each animation will have an add button for adding each individual sprite in the animation.
FPS	Frames per second. The rate at which the animation plays.
Loop	If this is enabled, the animation will loop. This could be useful for simple talking animations.

A dialogue is a series of conversations. Create as many as required. Each conversation will have a series of messages. Each message can either be of type Message or Choice. If type Choice is enabled, it should be the last message in the conversation because the player will be asked to make a choice, and each option in the choice will branch into another conversation. The dialogue will end once the current conversation reaches the final message that is of type Message.

Property	
Conversation Bar	This displays the name of the conversation. From here, delete the conversation if necessary, add a new message, or open the box for more fields.
Name	Conversation name.
Is Random	If enabled, this conversation will be treated as a random conversation.
Message Bar	This displays the messenger's icon. Choose which messenger is talking and the type of message. From here, delete the message if necessary, or open the box for more fields.
Animation	If enabled, type the name of the animation that needs to play during the conversation.
Message To	If Position Icon is enabled, specify who the message is being delivered to so the system can know how to position the UI icon.
On Message Complete	The Unity Event invoked after the message is finished loading.
Message Type	If enabled, a standard message will be displayed. Type the desired message in the text box.
Choice Type	If enabled, press the red button to create a series of options for the player to choose from. Typically, the option response should be as short as possible, since the options will be mapped to UI button elements.
Branch To	If the player chooses this option, this will become the next conversation.
On Choice	If the player chooses this option, this will be the Unity Event that is invoked.

Note

When dialogue is started, the first conversation in the list will be the entry point.

DialogueUI

The system implements Unity's UI system to display the dialogue box. The UI elements in the dialogue box can be configured to your personal taste. The important part is to set the UI references so that the system knows when to enable them and set their content.

The demo contains a basic UI dialogue structure, which can be used as a starting point for other designs, but in general, the system will be looking for five basic elements: an icon image, a background image, a next signal image,

a TextMeshPro for the message itself and the messenger's name. If any UI elements are missing, the system will simply ignore them. In reality, only one of these is truly needed, and that is the message element. Thus the dialogue box can be configured as complex or as simple as necessary.

References	
Icon	The image of the messenger's avatar (or animation).
Background	The image background for the entire message. This image can be personalized for each messenger in the Dialogue component. This UI element should be considered the main component of the dialogue box. All other UI elements should be children to it.
Next Signal	The image that will appear once the message has been completely loaded, signaling the user to continue with the conversation. Implement a bouncing effect with LetsWiggle.
Message	The TextMeshPro of the message.
Messenger Name	The TextMeshPro of the messenger's name.
Choices	The buttons mapped to a choice in the conversation. Create the required amount. These buttons will be clicked or pressed by the user.

The following Unity Events are optional. If they're not used, the message will load immediately. But if used, they will be able to add special effects to any dialogue. It is, however, very important to connect them properly, or the dialogue system will not work correctly.

The system comes included with two very useful classes that will add effects to the dialogue box. **LetsWiggle** is a tweening library that can be used in the inspector to setup tweens, like moving and scaling UI. **TextMeshProEffects** will add special effects, like a typewriting or wobble effect, to a message. It is these classes that can be used for OnTransitionIn, OnTransitionOut, and On Load Message. These classes should be placed on the background UI Element.

Property	
On Begin	The Unity Event that is invoked when a dialogue begins.
On End	The Unity Event that is invoked when a dialogue ends.
On Transition In	Invoke this Unity Event when a new message is about to begin. How it's intended to work: This event should call another class, like a tweening library, to move or scale the background image. In this case, since it's a transition In, the background image can be scaled from a 0 to 1. Once complete, it is very *important* for this other class to call **TransitionInComplete** in DialogueUI. If this event is used, there should *always* be a return signal to let the system know it can move forward, or else the system will stay stuck.
On Transition Out	Invoke this Unity Event when a message ends. This works exactly as On Transition In, but in reverse. The important part is for the other class to call **TransitionOutComplete** in DialogueUI once complete.
On Load Message	Once a Transition In is complete, this Unity Event will be invoked. Typically you call a class that will add special effects to the message itself. As with the previous two events, once the other class is complete, it must return a complete signal by calling **MessageLoadingComplete** in DialogueUI. If it does not, the system will get stuck.

Warning

If using OnTransitionIn, OnTransitionOut, or OnLoadMessage, it is very important to set the Unity Events properly or the system will get stuck.

Equipment

The Equipment class keeps track of all the tools the character has. A tool is simply a child gameobject of a character that contains a component that inherits from Tool. This class belongs to the Character class.

As of right now, the only tool the system implements is the Firearm class, but it's possible to create more tools as long as the new tool inherits from the Tool class. Call the following methods to activate or deactivate tools (firearms).

Method	
ActivateTool (string toolName)	Activate this tool.
ActivateThisToolOnly (string toolName)	Activate this tool and turn off all others.
DeactivateTool (string toolName)	Deactivate this tool.
DeactivateAllTools ( )	Deactivates all tools.
DeactivateAllToolsExcept (string toolName)	Deactivates all tools except this tool.
EquipmentIsActive ( )	Returns true if more than one tool is active.
RegisterTool(Tool tool)	Register the following tool into the system.
RemoveTool(Tool tool)	Remove the following tool from the system.
ToggleTool (string toolName)	Toggle the active state of this tool.
ToggleThisToolOnly (string toolName)	Toggle the active state of this tool and turn off all others.
ToolIsActive (string toolName)	Returns true if this tool is active.
ToggleOrActivateOnly(bool toggleTool, string toolName)	If toggleTool is enabled, the activate state of the tool will be toggled. If it's false, the tool will activate and turn off all others. This is used by the inventory system.
ToggleOrActivate(bool toggleTool, string toolName)	If toggleTool is enabled, the activate state of the tool will be toggled. If it's false, the tool will activate. This is used by the inventory system.
ChangeFirearmProjectile (ItemEventData itemEventData)	This is called by the inventory system to change the projectile of the first active tool(firearm).
ResetAll()	Calls the ResetAll method for each tool.

Firearm

The Firearm shoots projectiles. The purpose of this component is to control the behavior of the weapon as it relates to the character. It is not concerned about how projectiles operate. You will specify its fire point, the projectile it uses, shooting direction, how it rotates, recoil, and other features. It's possible to create many variations of a firearm with these properties.

To set up, create an empty gameobject and add the Firearm component. Add a Sprite to this gameobject if the weapon should be visible (and set Sprite Pivot to Left). Make this gameobject a child of the character that will be using the firearm. The player must have the Firearms ability enabled (for recoil to work and for applying ability exceptions).

Note

More than one firearm can be active at a time.

Property	
Name	Give the firearm a unique name for identification purposes.

Fire Point	The spawn point for projectiles. Create a transform for the fire point and make it a child of the firearm. Set the reference here. The fire point is typically placed at the end of the muzzle.
Button	This the shoot button. Choose the button and its trigger event.
Local Position	The local position of the firearm.
Off Near Wall	If the character is next to a wall and the weapon is inside the wall, setting this true will prevent the weapon from shooting. This will ensure the weapon cannot shoot into another room.
On Activated	The Unity Event invoked when the firearm's gameobject is set active true.

Method	
Shoot()	If necessary, trigger a shooting event with this method.
AnimationComplete()	Call this method when the shooting animation completes. This will not trigger a shooting event, but it will terminated the animation state of the firearm. It is assumed that ShootAndWaitForAnimation() was called before. If using Sprite Engine, call this in the Loop Once Event.
ShootAndAnimationComplete()	Call this method when the sprite is complete. This will trigger a shooting event and terminate the animation state of the firearm. If using Sprite Engine, call this in the Loop Once Event.
ShootAndWaitForAnimation()	Call this method at the exact sprite frame where shooting is required. This will not end the animation state of the firearm. Therefore, the last frame in the shooting animation should call AnimationComplete();
ChangeDefaultProjectile (ProjectileBase newProjectile)	Change the projectile type.
ChangeChargeProjectile (ProjectileBase newProjectile)	Change the charge projectile type.

Note

The animation methods are only necessary if Shoot Animation is enabled. If Shoot Animation is enabled, the firearm will enter an animation wait state upon shooting, where the system will wait for the animation to complete.

Projectile

Property	
Default Projectile	The projectile the firearm will be using.
Inventory (Optional)	The reference to Projectile Inventory. This contains a list of projectiles, which makes it simple for the system to swap projectiles on the firearm.
Auto Discharge	After the character is finished shooting, specify a small duration in which the firearm keeps shooting. If the value is zero, this feature will be disabled.

Shoot Animation	If enabled, it's the animation the character plays before shooting. The system will set the specified animation signal true. If using Sprite Engine, make sure the signal name exists, configure the Sprite State accordingly, and set to Loop Once. If this is enabled, **a return signal is required or the system will stay stuck.** Typically you will call ShootAndAnimationComplete (), or ShootAndWaitForAnimation () followed by AnimationComplete ();

Property	Recoil will provide pushback upon shooting a firearm.
Type	If Slide Back is enabled, the player will slide back. If Shake is enabled, the player will slide back and then slide forward. Shake is meant for short time durations.
Recoil When	If Velocity Zero is enabled, recoil will only work if the player is standing still.
Distance	The distance the player recoils.
Time	The time required to recoil.
Has Gravity	If the player is influenced by gravity, enable this so the recoil system takes the gravity effect into consideration.

Signals: recoil, recoilLeft, recoilRight, recoilDown, recoilUp, recoilSlide, recoilShake

Important

The player must have the Firearms ability enabled for recoil to work.

Note

Since recoil will apply a velocity to the character, the direction of the sprite will flip. If using Sprite Engine, to keep the sprite in the correct state, enable recoil, recoilLeft, and recoilRight signals, and configure the state of the sprite direction. Typically, if recoilLeft is true, set flipLeft, and if recoilRight is true, set flipRight.

Rotation

Rotate the firearm.

Property	
Rotate With	How to rotate the firearm: Mouse, Keyboard, Auto Seek, Fixed Angle, No Rotation.
Point Towards	The direction the firearm will point towards. If Character Direction is enabled, the firearm will point in the direction of the character movement. If Mouse Direction is enabled, the firearm will point in the direction of the mouse. If Transform Direction is enabled, the firearm will point in the direction of the specified Transform. This can be useful in case the firearm belongs to an npc; the specified transform can be set to point towards the player.

Signals: mouseDirectionLeft, mouseDirectionRight

Note

If using Sprite Engine, enable the mouse signals and use them to keep the player facing the direction of the mouse, even if the player is running in the opposite direction. Typically, if mouseDirectionLeft is true, set flipLeft, and if mouseDirectionRight is true, set flipRight. The character's running sprite might need extra consideration, as it will probably need to play in reverse to achieve a running backwards look.

Property	Rotate With Mouse
Top Limit	The range of motion of the firearm from 0 to 180 degrees. This will only be enabled if Point Towards is set to Character Direction.
Bottom Limit	The range of motion of the firearm from 0 to -180 degrees.
Angle Offset	If desired, the offset that is applied to the firearm.

Note

If Top Limit and Bottom Limit both have zero values, the range of motion of the firearm will be 360 degrees. Otherwise, the firearm is considered clamped, and it will be restricted by the specified limits.

Property	Rotate With Keyboard
Buttons Left, Right, Up, Down	Specify the keyboard buttons for changing the direction of the firearm.
Diagonal	If enabled, the firearm will be able to point in 45 degree angles.

Property	Rotate With Auto Seek (Rotate Towards Targets Automatically)
Target Layer	The layer where targets should be searched for.
Search Radius	Only targets within this radius from the center of the firearm will be detected.
Search Rate	How often the firearm should search for targets.
Auto Shoot	If enabled, the firearm will automatically shoot at a target.
On Found New Target	The Unity Event invoked each time the firearm finds a new target.

Aim

Add visual elements, such as a beam or reticle, to help the player aim.

Property	Beam
Line Of Sight Beam	A beam will extend from the firearm. Create a gameobject and place a sprite that represents the beam. Set the Sprite Pivot to Left. This gameobject should be a child of the firearm, and it should be enabled.
Layer	The layers the beam can interact with. Typically these are walls and enemy targets.
Beam	Place a reference of the beam gameobject here.
Beam End	This is optional. If the beam hits the target layer, this gameobject will be enabled at the end of the beam. It can play a sprite for special effect purposes. Create this gameobject and make it a child of the firearm. This gameobject should not be enabled. The system will control its active state.
Max Length	The max length of the beam. If no target layer is hit, the beam will extend to this length.
Target	The layer that should contain only enemies. If the beam hits an enemy target, the On Target Hit event will be invoked.
Auto Shoot	If enabled and an enemy target is hit using the Target layer, the firearm will shoot automatically.
On Target Hit	The Unity Event invoked when an enemy is hit using the Target layer.
On Beam Hit	The Unity Event invoked when the beam hits anything.
On Nothing Hit	The Unity Event invoked when the beam hits nothing.

Property	Reticle
Line Of Sight Reticle	A reticle will be displayed at a specific distance from the firearm. Create a gameobject and place a sprite that will act as the reticle. Make this gameobject a child of the firearm.
Aim Reticle	Place a reference of the reticle gameobject here.
Follow Type	If Fixed Position is enabled, the reticle will point in the direction of the firearm at the specified Distance. If Follow Mouse is enabled, the reticle will follow the exact position of the mouse.
Distance	If Fixed Position is enabled, specify how far the reticle should be from the firearm.

Charge

Charge the firearm to unleash a super charged bullet attack.

Property	
Projectile	The projectile used in the discharge attack.
Charge Time	The time the user must hold a button before triggering a discharge. It is also possible to set a minimum threshold time in which the player can still discharge the firearm.
Discharge Time	The time required to discharge the firearm completely.
Cooldown Time	The time that must elapse before the next charge can begin.
Shoot Animation	If enabled, it's the animation the character plays before discharging. The system will set the specified animation signal true. If using Sprite Engine, make sure the signal name exists, configure the Sprite State accordingly, and set to Loop Once. If this is enabled, **a return signal is required or the system will stay stuck.** For a charging setup, only call ShootAndAnimationComplete () once the animation is complete.

Event	
OnCharging	The Unity Event invoked when the firearm is charging.
OnCharging Complete	The Unity Event invoked when the firearm finishes charging.
OnDischarging	The Unity Event invoked when the firearm is discharging. This will return a value between 0 and 1, representing the remaining discharge time.
OnDischarging Complete	The Unity Event invoked when the firearm finishes discharging.
OnDischarging Failed	The Unity Event invoked when the firearm fails to discharge. This can result from not having enough ammo.
OnCooling Down	The Unity Event invoked when the firearm is cooling down. This will return a value between 0 and 1, representing the remaining cool down time.

Recoil	
On Discharge	This recoil has the same properties as the default projectile recoil. While Discharging, specify if the player should recoil only once or as many times as required during the discharge period.

Note

The charge system requires the user to hold a button for charging. Thus, it's best to set the trigger type of the normal shoot button to On Press. This way, the fire arm won't be shooting bullets while charging. Also, the projectile used for charging should be set with a high fire and projectile rate to take advantage of the continuous discharge.

Inventory

The inventory system allows the player to find and store items, provides a customizable user interface for interacting with the items, and automatically restores and saves the items when a scene begins and ends. The system itself relies heavily on Scriptable Objects, which allow for a flexible foundation.

There are five main components: **Item**, **PickUpItems**, **Inventory**, **InventorySO**, and **ItemSO**. The player, using PickUpItems, will pick up an Item and add it to InventorySO. The Item is then mapped to an ItemSO, which contains the necessary information to display in the Inventory. From there, the user can pick, move, and use items.

The UI structure in Inventory simply displays the contents of InventorySO. It's for this reason that multiple inventories can be created to work together. Thus, it's possible to have an inventory only for firearm items and another for all the items in the system.

InventorySO

This Scriptable Object is the glue that holds the entire inventory system together. Its primary job is to contain a list of all the items, to add and remove items, and to save these items in memory. Each InventorySO must have a unique name for saving purposes. Also, for it to function properly, it must be referenced by at least one Inventory.

It's possible to have multiple inventories working together, all referencing the same items. To accomplish this, those inventories must reference the same InventorySO, and that's it.

To create an InventorySO, right click in the project window and go to Create/FlareEngine/InventorySO.

Property	
Reference Inventory	Any item that can potentially exist in the inventory should have its corresponding ItemSO added to the Reference Inventory. The character will not be able to pick up an item if the item's reference ItemSO is not part of this list.
Default Items	Any ItemSO in this list will be immediately inserted into the active inventory. Use this carefully. The system will *always* try to add these items if they don't exist in the inventory. For example, if you default a health item, this item will be added each time the inventory lacks one. This is more suitable for non-consumable items, like setting a default firearm.

Warning

A common source for odd behavior is simply not updating the Reference Inventory each time a new ItemSO is created.

Method	
Save( )	This will save the inventory in memory. It's called by OnDisable when the Scene terminates. But you can call this method anytime you wish. You can also use the static method **InventorySO.SaveData ( )**, which will save *all* the inventories. The **using TwoBitMachines** namespace will be required.

ItemSO

This Scriptable Object is a template for an item the player can pick up. It contains the necessary information and settings for handling the life cycle of the item.

To create an ItemSO, right click in the project window and go to Create/FlareEngine/ItemSO.

Property	
Item Name	The name of the item. This must be unique for identification purposes.
Key Name	Once an item in the inventory is used, PickUpItems will use the Key Name to find the correct Unity Event to trigger.
Icon	The item's icon. This sprite will be displayed in the inventory.
For Inventory	If AddToInventory is enabled, the item will be added to an inventory. Otherwise, the item will be used immediately. If the item cannot be used immediately (maybe the value has already reached a max value), the system will try to add it to the inventory as a backup measure.
Droppable	If No is enabled, the item cannot be dropped from the inventory. If Yes is enabled, it can be dropped, and a prefab of the item can be specified. This prefab will be instantiated in the scene next to the character. If no prefab exists, the item will still be removed from the inventory.
Consumable	If enabled, the inventory removes this item when it's used (think coins or health). Consumables of the same type can be stacked in the inventory. Set the limit of the stack. If the limit is reached, the inventory will simply create another slot item and begin the stacking process again. If this is not enabled, the item will remain in the inventory if used. Think firearms. In this situation, there can only ever be one of these items in the inventory. It wouldn't make sense to have two of the same firearms.
Generic Float And String	The meaning of these values is completely arbitrary. They may not even be necessary at all. Their purpose relies entirely on what the item needs to execute. For example, when a firearm item is used, the corresponding event will simply activate or deactivate the gameobject of the real firearm, never even using these values. If it's a health item, only the Generic Float is required to specify the amount of health that will be applied to a character.
Cost	If the inventory this item belongs to is of type Vendor, this will determine the cost of the item. Also choose whether to keep or remove this item in the inventory upon selling it.
Item Description	Describe what the item does. This is displayed in the inventory.

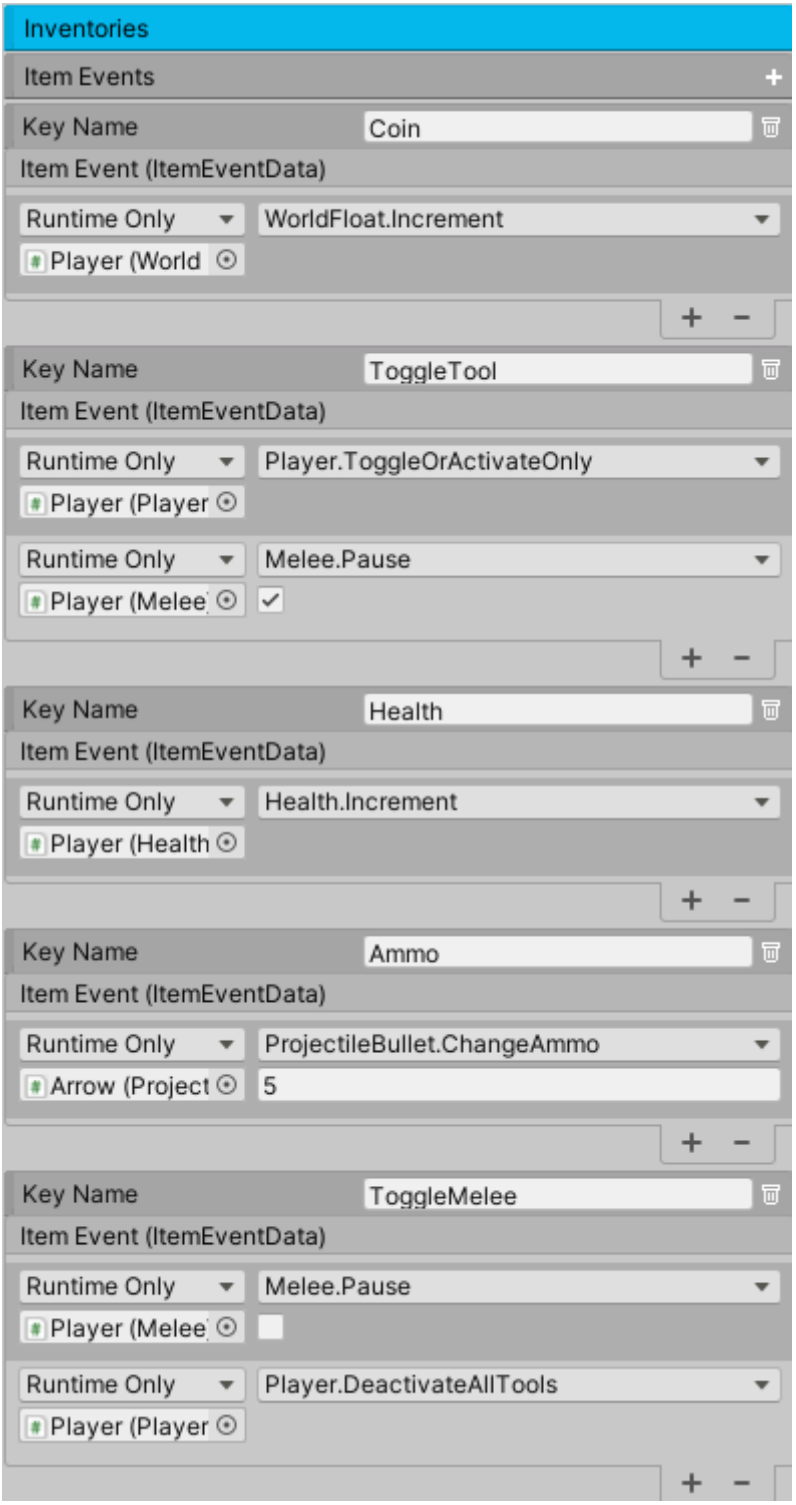
PickUpItems

Attach this component to a character. Upon contact with an item, this will deactivate the item and add it into the inventory. If this item is not a consumable, and it already exists in the inventory, the system will ignore it.

Property	
Inventories	Set a reference for each Inventory the character is using. PickUpItems will initialize these inventories during Awake for proper function.
Item Events	Each item has a Key Name (found in ItemSO) which belongs to a corresponding Item Event. When an item is used, the system will find and trigger the correct Item Event and execute the intended purpose of the item. This Event is invoked with the argument ItemEventData, which is a class that contains the item's generic values and a success boolean. Any method called by Item Events must have ItemEventData as a parameter.

Is Vendor	If enabled, this means the inventory does not belong to the player. The player must purchase items from this inventory using money. Money is represented by a WorldFloat component, which can represent coins. Coins will typically be items the player must collect in the game world. Set the references to the player's transform, which will be used to retrieve the player's PickUpItems component, and a WorldFloat, which will be used to buy items. There are two events - one for a successful transaction and one for a failed transaction (not enough money). The vendor can be controlled by the AI system to trigger the inventory UI menu.
Mouse Pick Up	If enabled, the user can add items into the inventory by clicking on them. Specify the layer where the items exist.

It's important to realize the values in ItemEventData are set internally. The only value to be concerned with, if implementing your own methods, is to set itemEventData.success to true or false so the inventory system can properly handle the item once it's used. Let's look at some examples of Item Event.



If you want to use a health item, connect the Item Event to the Health Component and choose the Dynamic method Increment. That's it. Internally, the system will use the item's generic float value to increase the health. If the Increment method succeeds, it will set itemEventData.success to true. If set to false, the system will interpret the event as a failure and retain the health item in the inventory.

Very similarly, if you want to use a firearm item, connect the Item Event to the Player Component and choose the Dynamic method ToggleOrActivateOnly. That's it. Internally, the system will try to toggle the active state of the firearm gameObject and turn off all other firearms (if any). This method will automatically set itemEventData.success to true, since in this case the item is not a consumable. The other values of ItemEventData are not used.

If you want to change the projectile of a firearm using an item, connect the Item Event to the Player and choose the Dynamic method ChangeFirearmProjectile. For this to work, the Firearm component must contain a ProjectileInventory (which contains a list of projectiles). This method will use the generic string value, which specifies

the name of the new projectile. The system will then set ItemEventData.success to true if it succeeded in changing the projectile. It's important to note this method will change the projectile of the first active firearm the system finds. If you want a more specific approach, instead of using Player, use the Firearm component.

To increase a projectile's ammo using an item, connect the Item Event to a ProjectileInventory and choose the Dynamic method ChangeProjectileAmmo. In this case, the itemEventData will use both generic values. It will use the generic string to specify the name of the projectile it wants to modify, and it will use the generic float value to change the ammo amount. As always, itemEventData.success will bet set.

Item

Attach this component to any gameobject that is an item. This item will become part of the inventory (and its gameobject will be deactivated) when a character makes contact with it. Attach a Circle Collider2D and set the isTrigger to true. Each item must correspond to an **ItemSO**.

Property	
ItemSO	The reference to ItemSO.

Inventory

The Inventory contains the UI structure. Place this component on a Canvas gameobject. The slots (item containers) and accompanying features can be arranged however you deem necessary. You can arrange the slots in a grid, in a vertical or horizontal bar, or you can have just one slot. It's completely up to you.

Slots are the most important elements of the Inventory. Everything else is optional. Each slot must contain the **InventorySlot** component.

Property	
InventorySO	The reference to InventorySO.
View Items	If All is enabled, all the items in the inventory will be displayed in this inventory. If Key Name is enabled, only items with the specified Key Name will be shown. This can be great for creating an inventory that only contains firearms.
Navigation	If Unity Navigation is enabled, the keyboard/mouse will be the primary method for navigating the slots. Remember to set Send Navigation Events (in Event System) to true, and any UI Button not part of the system should have its Navigation set to None. This setting is more appropriate for slots arranged in a grid. If set to Manual, buttons will be used to *move the items* across the slots. Use this option if the slots are arranged in a bar.

Property	
Left, right	The buttons for moving the items left and right, or up and down, depending on the layout of the slots.
Auto Item Load	If enabled, the item in the first slot will be used automatically.
Toggle Inventory	A button for toggling the active state of the Inventory gameobject. If no button is desired, set to None. If the button exists, specify if the game should pause when the Inventory is activated. Pausing the game will also block player input. If Block Player Input is enabled, only the player input will be paused when the inventory is open.
Use Item - Slots	If OnSlotSelection is enabled, when the user clicks or presses a slot, the item will be used immediately. If not enabled, the item is only selected but not used.
Use Item	A button for using items. If Navigation is set to Unity Navigation, the selected item will be used if this button is pressed. If Navigation is set to Manual, then the item in the first slot will be used. If no button is desired, set to None.

UI references are all optional and can be placed wherever necessary. If used, the system will be in charge of setting and enabling them when appropriate.

Property	
Item Name	The TextMeshPro displaying the name of the selected item.
Item Selected	The UI Image displaying the icon of the selected item.
Item Description	The TextMeshPro displaying the description of the selected item.
Item Cost	The TextMeshPro displaying the cost of the selected item. This is used for an inventory that is of type vendor.
Current Item	The UI Image displaying the icon of the current non-consumable item. This UI Image usually exists outside the inventory to remind the player of the current item.
Drag Item	The UI Image with a DragItem Component. If it exists, it will allow items to be dragged and swapped over slots. The item's icon will appear next to the mouse as the item is being dragged. If the slots are using a Unity Grid Layout, attach a Layout Element component to this gameobject and set IgnoreLayout to true. This will allow the UI Image to move freely.
Remove Item	The UI button reference for removing items. If an item can be dropped, this element will become visible.
Select Frame	The UI Image that will automatically move to the selected slot and highlight it. The speed specifies how quickly it moves to a new slot. If set to zero, the image will be instantly moved.
Refresh Slots	This button must be pressed each time inventory slots are added or removed from the UI structure.
Delete Slots	For testing. If pressed, this will delete all the saved data in the inventory.

Events	
OnOpen	The Unity Event invoked when the inventory is opened.
OnClose	The Unity Event invoked when the inventory is closed.
OnMove	The Unity Event invoked when the user clicks a new slot.
OnUseItem	The Unity Event invoked when an item is used.
OnRemoveItem	The Unity Event invoked when an item is removed.

InventorySlot

Inventory Slot is used by Inventory for displaying items. Each slot requires this component. The demo includes the proper setup, including connections for dragging and swapping items.

Property	
Image Icon	Displays the item's icon.
Image Drop (Optional)	If the UI Image exists, it will be enabled on item's that are droppable. The button belonging to this image should call the DropItem method to drop the item back into the scene.
Text Mesh (Optional)	If the TextMeshPro exists, it will display the item's stack count if the item is a consumable.

Method	
UseItem ( )	Use the item in this slot.
DropItem ( )	This will instantiate the item in this slot back into the scene next to the character.

Lets Wiggle

This is a tweening library that is configured in the inspector or in code. The **Wiggle** component, which is different from the **LetsWiggle** component, must be included in the scene for proper function. The **World Manager** component will automatically include it for you.

LetsWiggle allows you to create a series of tweens in the inspector. These tweens are chained and execute in series or in parallel. Each tween belongs to one of three categories: Move, Scale, or Rotate. Thus, each tween has a specific job to execute, like moving or scaling a Transform. For example, Move2D will move a Transform (or Rect Transform) in 2D space.

Some tweens include the word **To** and some don't. This means, for example, MoveX will simply move in the x direction, while MoveToX will move to a specific position in the x direction.

Most tweens will have a corresponding tween function. This is where the magic happens. A tween function tells the tween *how* to move. Create as many tweens as necessary to accomplish the desired effect.

List Of Tweens:

- Move2D
- Move3D
- MoveX
- MoveY
- MoveZ
- MoveTo2D
- MoveTo3D
- MoveToX
- MoveToY
- MoveToZ
- ScaleTo2D
- ScaleTo3D
- ScaleToX
- ScaleToY
- ScaleToZ
- RotateTo
- RotateX
- RotateY
- RotateZ
- RotateAroundX
- RotateAroundY
- RotateAroundZ

List of Tween Functions:

- EaseIn
- EaseOut
- EaseInOut

- EaseInBack
- EaseOutBack
- EaseInOutBack
- EaseInBounce
- EaseOutBounce
- EaseInOutBounce
- EaseInCubic
- EaseOutCubic
- EaseInOutCubic
- EaseInElastic
- EaseOutElastic
- EaseInOutElastic
- EaseInExpo
- EaseOutExpo
- EaseInOutExpo
- EaseInSine
- EaseOutSine
- EaseInOutSine
- Spike
- Gravity
- GravityBounce
- GravitySingleBounce
- OnOff
- Linear

Settings

Property	
Name	Name the wiggle.
Target	The gameobject that will be tweened. Set the reference here. The system will determine if the gameobject belongs to a Transform or Rect Transform.
Loop All	Once all the tweens have executed, loop all of them by the specified amount.
Unscaled Time	If enabled, the system will use unscaledDeltaTime instead of deltaTime.
Use Anchors	If the gameobject is a Rect Transform, choose to move its position by anchoredPosition or position.
Start Position	If enabled, reset the Transform to this position before executing the first tween.
Start Rotation	If enabled, reset the Transform to this rotation before executing the first tween.
Start Scale	If enabled, reset the Transform to this scale before executing the first tween.
Start On Awake	If enabled, tweening execution will begin on Awake.
Start On Enable	If enabled, tweening execution will begin OnEnable.

Off On Complete	If enabled, the gameobject will deactivate once the tweening is complete.

Method	
Wiggle()	Start the tweening execution.
PauseWiggle(bool value)	Pause or unpause the tweening.

Wiggle Bar

Open the bar and start adding tweens. Each tween will have at least a couple of fields to set, and all of them will share the properties below.

Property	
Delay	The wait time before the tween starts.
Loop	The number of times this specific tween will loop.
Parallel	If enabled, this tween will execute in parallel with adjacent tweens that also have Parallel enabled. If not enabled, the tween will execute in series.
On Complete	The Unity Even invoked when the tween is complete.

Code

The Wiggle library is also accessible through static methods using the **TwoBitMachines** namespace. Below is an example of how to use the tweening library. It's just a very long line of C# code. To get started, implement the **Wiggle.Target** static method to start chaining tweens.

Tip

To deactivate the gameobject on tweening completion call the Deactivate method.

```
Wiggle.Target (this.gameObject, useAnchors: false).
StartPosition (Vector3.zero).
Move2D (Vector2.up, 2f, Tween.Linear).Loop (2).
ScaleToX (0.5f, 2f, Tween.EaseIn).IsParallel ( ). // Scale x to 0.5f in 2 seconds
ScaleToY (0.5f, 2f, Tween.EaseOut).IsParallel ( ).
ScaleToX (1f, 1f, Tween.EaseInBounce).
ScaleToY (1f, 1f, Tween.EaseInBounce).
LoopAll (2);                                    // Can also call Deactivate() here
```

Important

The methods below will only work on tweens created in code.

Method	
IsTweenActive (GameObject gameobject)	Returns true if this gameobject has an active tween.

PauseTween (GameObject gameobject)	Pause tweening on this gameobject.
UnpauseTween (GameObject gameobject)	Resume tweening on this gameobject.
StopTween (GameObject gameobject)	Stop tweening on this gameobject.
PauseAllTweens ()	This will pause all active tweens.
StopAllTweens ()	This will stop all active tweens.

Melee

Create a melee attack for the player to use against enemies. Each melee attack consists of a collider that will be used to detect targets and an attack animation. It is possible to chain attacks based on user timing and hit success. It is also possible to add velocity to each attack.

However, the attack animation has to be setup and executed elsewhere. If using Sprite Engine, set the attack animation to loop once, and add a collider2D property to control the size and position of the collider on a per frame basis. The Melee system will be in charge of enabling the collider, and setting the animation signal active. Once the attack animation is complete, the **CompleteAttack** method must be called on the **Melee** class.

To create a melee, create a gameobject and set it as a child of the player transform. Add the collider that will be used to deal damage. Then add the Melee component. The player must have the Melee ability enabled. Once enabled, register the newly created melee attack. Since more than one melee attack can be registered, it is thus possible to change between melee attacks.

Melee	
Melee Name	Name the melee for identification purposes.
Hit Layer	The layer the collider2D will check for collisions.
Melee Collider	The reference to the collider2D.
Button	The button and its trigger type that will start a melee attack.

Properties	Some fields are only enabled if Combos has more than one attack.
Hit To Continue	If enabled, in order to move on to the next attack in the combo, the current attack must have made contact with a target.
Early timer	How soon after the attack started can the user press the button in order to successfully begin the next attack in the combo.
Delay timer	How much time after the attack finished can the user still press the button in order to successfully begin the next attack in the combo.
Cool Down	The wait period before the next combo can begin.
On Cool Down	This event will be invoked with a percentage of the cool down time.

Combos	
Sprite Name	The animation signal the system will set true when performing a combo.
Damage	The amount of damage dealt to the target hit.
Velocity x	The velocity applied to the combo in the x-direction.
Velocity y	The velocity applied to the combo in the y-direction. If Jump Velocity is enabled,the velocity will be treated as a jump force.
On Combo Begin	The event invoked when the Combo begins.

Signals: meleeCombo

Projectiles

Projectiles are for shooting. There are three basic types: Bullet, Instant, and Short Range, and each once can be customized to create plenty of variety. Each type will deal damage to any gameobject that has a Health component and exists in the target layer of the projectile.

In general, the Projectile component should exist in a gameobject that has a static parent to preserve the local positions of the projectiles. The Projectile component will create an object pool and manage the life cycle of all its projectiles.

Projectile

Property	
Name.	Name the projectile. This must be unique for identification purposes.
Projectile Type.	A Bullet is a discrete gameobject. An Instant is a raycast. Short Range is a BoxCollider2D.
Ammo Type	If Discrete is enabled, the ammo count will be decreased by a unit of one. If Continuous is enabled, the ammo will be decreased by a unit of time. Typically, a Bullet is set to Discrete, Short Range is set to Continuous, and Instant can be either depending on the setup. If Infinite is enabled, ammo will never run out.
Ammo Amount	The available ammo for shooting. If Continuous is enabled and the value is 10, the projectile can provide the firearm with 10 seconds of shooting, for example. The max value clamps the ammo count.
Ammo Damage	The amount of damage dealt to any target with a Health component.
Fire Rate	How often a firearm can shoot this projectile. A value of zero will mean the firearm can shoot this projectile every frame.
On Ammo Reload	The Unity Event invoked when the ammo count is increased.
On Ammo Empty	The Unity Event invoked when the ammo count is zero.

Method	
ChangeAmmo(float value)	Increase or decrease the ammo amount.
AmmoCount()	Returns the ammo amount.
EnoughAmmo()	Returns true if the ammo amount is greater than zero.
AmmoIsInfinite()	Returns true if the ammo is infinite;
AmmoMax()	Returns the maximum ammo amount;
AmmoPercent()	Returns the percent of ammo remaining.

Bullet

There are six bullet types, each with a different behavior. There’s also no limit on how many firearms can reference a projectile type, making it great for saving resources.

Important

First create a gameobject that contains the bullet's sprite and make it a child of the Projectile gameobject; only then will the properties bar appear. **For proper function, set the Sprite Pivot to Left. If using bounce, set Pivot to Center.**

Property	Setup
Bullet Type	Basic, Bounce, Bounce4R, Colliding, Seeker, StickToWall.
Pool Size	The total number of bullets that can exist in the scene. The bullets will be created as needed and will never be destroyed.
Projectile Rate	The amount of bullets discharged during a single firing event. If this value is greater than one, then it's possible to create bullet patterns with the properties below.
Position Flux	This will vary the start position of a bullet in the y direction by the specified value.
Angle	The angle at which bullets will be fired relative to other bullets.
Separation	The relative space between bullets in the x and y direction.
Direction	If Weapon Direction is enabled, bullets will be fired according to the pattern settings. If Circular Direction is enabled, it will ignore those settings and shoot the bullets radially.

The bullet types share some of the following properties below. In the case of the Seeker type, it will ignore the gravity setting.

Property	
Layer	The target layer the bullet will interact with. The most common layers are enemies and walls.
Ignore Edges	Specify how to handle Edge Collider 2D collisions. If Ignore If Up Direction is enabled, a bullet will ignore this collision if the bullet has a positive y velocity.
Life Span	The amount of time the bullet can be active in the scene.
Speed	How quickly the bullet moves.
Gravity	If this value is greater than zero, the bullet will be affected by gravity.
Blast Radius	If this value is greater than zero, the bullet will deal damage to any target within this radius when the bullet collides or time expires.
Impact Effect	The name of the world effect that will be used on impact. If not needed, leave empty.
Bullet Rays	Bullets use only one raycast to detect targets. If more are needed, specify that number here.
Bullet Size	Specify the correct size of the bullet for proper bullet collision.
Add Momentum	If enabled, the character's velocity will be combined with the bullet's velocity.

Event	
On Fire	The Unity Event invoked when successfully fired.
On Impact	The Unity Event invoked when the bullet makes any kind of contact. This event can also invoke world effects. If the impact effect is specified, the hit point and direction will be applied to the world effect.
On Hit Target	The Unity Event invoked when the bullet hits a target that contains the Health Component. This event can also invoke world effects. If the impact effect is specified, the hit point and direction will be applied to the world effect.
On Life Span Expired	The Unity Event invoked when the bullet's time expires. The position of the bullet is returned.

Basic

This is the most basic bullet. It has no extra properties.

Bounce

This bullet will bounce off walls.

Property	Common
Bounce Friction	If this value is greater than zero, every time the bullet bounces off a wall, it will lose velocity.
Bounce Radius	The radius of the bullet. Required for collision checks.
Bounce Spin	How much the bullet spins on its axis. If the bullet is a bouncing ball (that means gravity is enabled), then set this to a nonzero value, or else the bullet will constantly rotate according to its direction, which might look inappropriate for a bouncing ball.

Important

Bounce and Bounce4R work exactly the same. However, Bounce4R uses four raycasts to detect walls. Use Bounce4R if more perfect collisions are necessary.

Colliding

This bullet uses a Collider2D instead of raycasts to detect targets.

Property	
Expire On Impact	If enabled, the bullet will not deactivate once it collides.

Important

Add a Collider2D (set Is Trigger true) and a RigidBody2D (set to Kinematic) to the bullet, or else there will be no collisions. The target layer should be used primarily for enemies and not wall collisions.

Seeker

This bullet will curve, change directions, to follow a target.

Property	
Search Radius	The radius around the bullet's position used for finding targets.
Search Rate	The bullet will execute a find function at the specified rate until it finds a target to latch to.
Turn Speed	How quickly the bullet can change direction.

Find	If Random Target is enabled, and if more than one target is found, the bullet will pick a random target from the list to follow. If Nearest Target is enabled, the bullet will follow the nearest target found.

Important

Since the Seeker bullet can take wide turns, the target layer should not contain walls or else the bullet will deactivate on a wall collision.

Stick To Wall

This bullet can stick to walls. Perfect for arrows!

Property	
Stick Timer	The amount of time the bullet sticks to the wall before deactivating.
OnStickToWallExpire	The Unity Event invoked when the bullet is done sticking to the wall. The bullet's position is returned.

Instant

A raycast is used to instantly hit a target.

Property	
Layer	The target layer the raycast will interact with. The most common layers are enemies and walls.
Ignore Edges	Specify how to handle Edge Collider 2D collisions. If Ignore If Up Direction is enabled, a raycast will ignore this collision if the raycast has a positive y velocity.
On Idle	If Deactivate GameObject is enabled, the gameobject will bet set Active false when the firearm is no longer shooting. This is useful in case this gameobject has a sprite that represents the raycast. If Leave As Is is enabled, it will leave this gameobject in its current state.
Max Length	The length of the raycast.
Hit Rate	If the firearm is fired continuously, set the rate at which a target can be hit. If left to zero, this means a target will be applied damage every frame. Avoid this to make the damage applied frame independent.
On Fire	The Unity Event invoked when successfully fired.
Impact Object	The gameobject that will be set active true at the impact point. Useful for particle effects for a laser. If not needed, leave empty.
Impact Effect	The name of the world effect that will be used on impact. If not needed, leave empty.
On Impact	The Unity Event invoked when the raycast makes a hit. This event is called at the same time as Hit Rate. This event can also invoke world effects. If the impact effect is specified, the hit point and direction will be applied to the world effect.

Important

It's possible to place a sprite on this projectile to act as a visual laser. Set the Sprite Pivot to Left to properly scale the sprite image from firearm to hit point.

Short Range

A BoxCollider2D will search for targets to hit.

Property	
Layer	The target layer the BoxCollider2D will interact with.
Collider	Place a BoxCollider2D component on this gameobject, configure the size, then set the reference here
On Idle	If Deactivate GameObject is enabled, the gameobject will bet set Active false when the firearm is no longer shooting. If Leave As Is is enabled, it will leave this gameobject in its current state.
Hit Rate	If the firearm is fired continuously, set the rate at which a target can be hit. If left to zero, this means a target will be applied damage every frame. Avoid this to make the damage applied frame independent.
On Fire	The Unity Event invoked when successfully fired.

Tip

Place a sprite to go along with the BoxCollider2D. There's also a Flame Thrower component made specifically for this projectile type. Place it on this gameobject and configure the particle properties to shoot some flames!

Projectile Inventory

This holds a list of projectiles. The inventory system uses this list to swap projectiles on a firearm. Create as many as necessary since firearms can reference different projectile inventories.

Property	
Projectile References	The list of projectile references. Create as many as necessary.

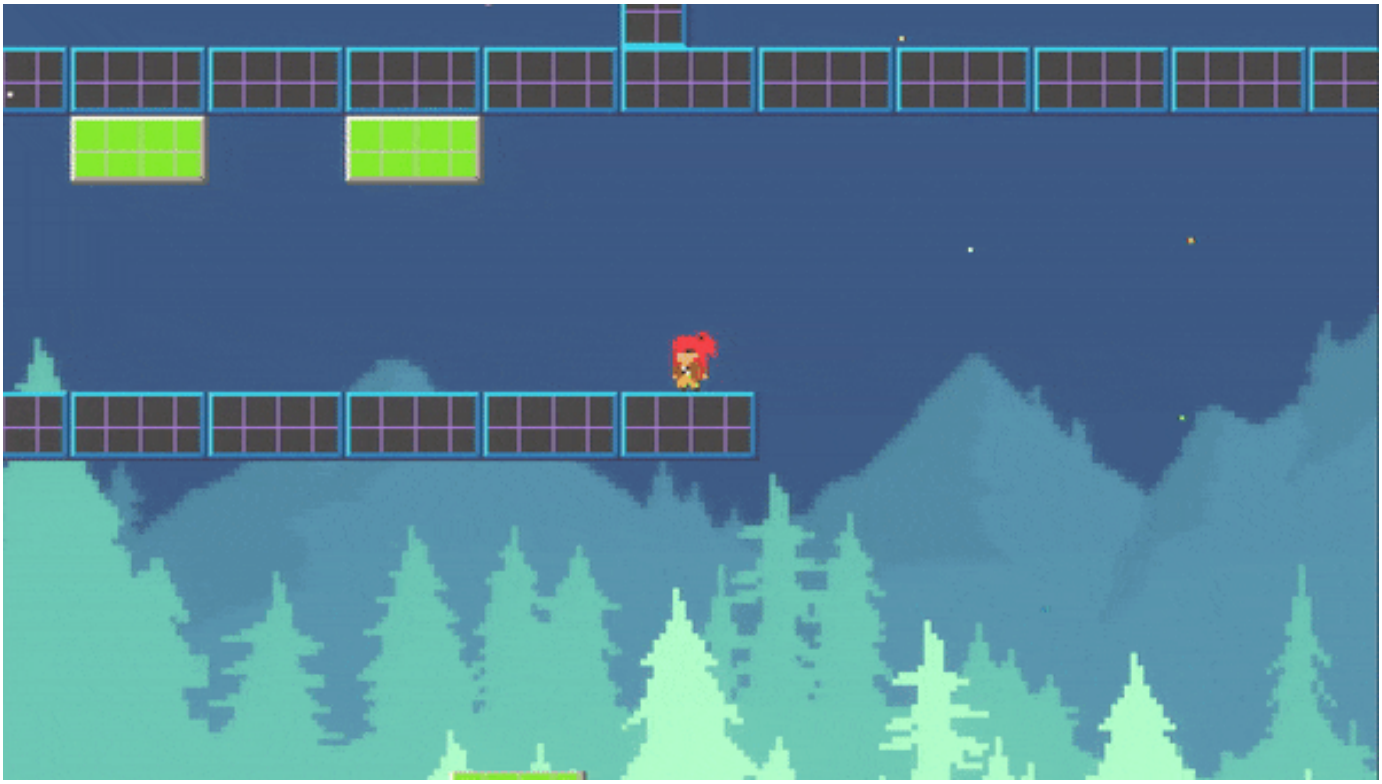
Scene Management

Transition from the current scene to the next scene with ease. The scene manager loads the specified scene and controls the screen transitions, if any. The system comes equipped with plenty and ready to use screen transitions, which can be found in the AssetsFolder/Transitions/Prefabs folder. Simply drag the necessary prefabs into the Hierarchy and set their reference in the **ManageScenes** component.

Each transition gameobject consists of a canvas with an image that contains a Transition shader. The system implements the common technique of using a black and white gradient texture to specify the pattern of the transition. If desired, you can add your own gradients to a new material to create new transitions.

To get started, place a ManageScenes component on a gameobject, organize the scene transition gameobjects as children, set the references, and simply call the LoadNextScene() to load the next scene.

The system also comes with the **ScreenTransition** component, which is useful for displaying opening scene transitions and transitions for a game reset (where the game resets but doesn't change scenes).



Note

Click the add button to add transitions into the system. The types are TransitionIn, TransitionOut, and LoadScene.

Property	
Next Scene	If there are scenes in the build, specify which scene to transition to when LoadNextScene() is called.
Menu Scene	If there are scenes in the build, specify the menu scene to transition to when LoadMenuScene() is called.
Load Scene	If Automatically is enabled, once the scene has finished loading, it will transition completely into the new scene. If OnUserInput is enabled, once the scene is finished loading, the scene will wait for any user input before transitioning completely.
Pause Game	If Pause Game is enabled, the game will be paused during a scene transition.
Random Text	If more than one texture exists, the transition pattern will be randomized. Place textures here from the AssetsFolder/Transitions/Textures folder. Use the TransitionAny prefab if implementing this.
Transition In/Out.	This gameobject will be set active true if it exists. If this object is a transition, the transition pattern will reach complete coverage on Transition In. It will reach complete transparency on Transition Out. If Deactivate is enabled, this gameobject will be set active false when the Transition Time expires.
Load Scene	After the transitions complete, this step will finally begin to load the scene. The gameobject, if it exists, will be set active true. This object can be a loading scene (which is not a transition). By changing the load speed, this will slow down the loading. A value of 1 is maximum speed.

Events	
On Start	The Unity Event invoked when a Transition or Load Scene starts.
On Complete	The Unity Event invoked when a Transition or Load Scene completes.
Loading Progress Float	This Unity Event is invoked with a percentage of the loading time. This can be used to set UI elements.
Loading Progress String	This Unity Event is invoked with a percentage of the loading time in string format. This can be used to set UI elements.

Method	
LoadNextScene()	The next scene will be loaded.
LoadMenuScene()	The menu scene will be loaded.
LoadScene(string sceneName)	The specified scene will be loaded.

Screen Transition

Place the **ScreenTransition** component on a transition gameobject. This will primarily be used for the opening scene transition and for game reset transitions. If using it for an opening scene transition, make sure the gameobject is set active true so it executes automatically at the beginning of the scene.

If using it for a game reset, simply activate the gameobject for the transition to begin. When a transition completes, the gameobject will be automatically be set active false;

Property	

Type	If Transition In, the transition will reach maximum coverage. If Transition Out, the transition will reach complete transparency. If Both, the system will execute TransitionIn and TransitionOut in that order.
Time	The duration of the transition.
Reset Game	If enabled and type is Both, the game will be reset after Transition In completes and before Transition Out begins. Time scale will be set to zero during the first half, effectively pausing the game.
Random Text	If more than one texture exists, the transition pattern will be randomized. Place textures here from the AssetsFolder/Transitions/Textures folder. Use the TransitionAny prefab if implementing this.

Text Mesh Pro Effects

Create special effects for any TextMeshPro. This includes the classic typewriting effect and a variety of other effects that can be applied to individual words or sentences. Since the system is working with TextMeshPro, feel free to apply Rich Text Tags to your text. This system makes specif use of **<link>** for many of the effects below.

Property	
Text Mesh	The TextMeshPro that will have the effects applied to it.
Typewriter Speed	The speed at which letters are typed.
Typewriter Wobble	The wobble effect applied to the letters as they are typed.
Typewriter Fade	The fade effect applied to the letters as they are typed. The larger this number, the longer the fade effect will last.
Wobble	Apply a wobble like motion to words. To apply: **<link="wobble">This is a Wobble</link>**
Wave	Apply a wave like motion to words. Specify the speed, strength, and phase. To apply: **<link="wave">This is a Wave</link>**
Jitter	Jitter will have the effect of shaking the words. Specify the rate at which it occurs, and its strength. To apply: **<link="jitter">This is a Jitter</link>**
Distortion	This will distort the words. Specify the rate at which it occurs, and its strength. To apply: **<link="distortion">This is a Distortion</link>**
Pause	This will pause the typewriter for the specified time. To apply: **<link="pause1"></link> These words will not be typed until 1 second has elapsed.** To apply: **<link="pause0.5"></link> These words will not be typed until 0.5 seconds have elapsed.**
Speed	This will change the typewriter speed. To apply: **<link="speed3"></link> These words will be typed with a speed of 3.**

World Effects

The **WorldEffects** class is used for pooling effects. This is primarily used for displaying effects for bullet collisions. Since bullet collisions can occur frequently, the system will pool these effects and try to use them as efficiently as possible. Projectiles are equipped to call this class with a Unity Event that passes the ImpactPacket class.

You can place this component on the World Manager's gameobject. In the inspector, create a new element by clicking the add button. Name and set the gameobject reference of the effect. The effect can be anything, a particle system, a sprite, but whatever it is, it must have the ability to turn its gameobject to active false upon completion.

You can also use this class with WorldManger.get.effects (using TwoBitMachines.FlareEngine).

Method	

Activate (ImpactPacket impact)	Activates the effect specified by impact. Call this method using the projectile impact event.
ActivatePlusDirection (ImpactPacket impact)	Activates the effect specified by impact. This will rotate the prefab according to the direction of the projectile. Call this method using the projectile impact event.
Activate (string name, Vector3 position)	Activate the effect with this name at the provided position.
ActivatePlusDirection (string name, Vector3 position, Vector2 direction)	Activate the effect with this name at the provided position and direction.

World Variables

These components allow you to create data that can be easily referenced. This data is also automatically saved and restored during runtime to preserve important information. The three types of variables used are **Float**, **String**, and **Vector3**. The **Health** component is also part of this group, but it works as a Float underneath the hood.

Each of these variables can be linked to a corresponding scriptable object. For example, if using the Health component, the **WorldFloatSO** (scriptable object) that belongs to it can be used as a reference for UI elements to display the character's health. Any other system, like an enemy AI, can also read this value to change its behavior. This will help decouple your systems because information is exchanged between gameobjects without creating hard references.

Float And Health

Tip

Float can be treated as a boolean by setting its value to either 1 or 0 (True or False).

Property	
Name ID	The name of the variable. If using the save system, this name must be unique. Click the generate button to create a unique Name ID if necessary.
Value	The current value. This is the value that will be saved.
Clamp	Clamp the current value.
Broadcast Value	If enabled, and if the current value changes, this will also set the current value for all variables with the same Name ID.
Recovery Time	If this is Health, the amount of time after being damaged where the character can't take any damage.
Save	If enabled, the system will save and restore the current value.
Is Scriptable Object	If enabled, a button will appear to create a scriptable object (**WorldFloatSO**). If created, it will be found in the AssetsFolder/Variables folder. If the scriptable object is no longer required, delete it manually.

Events	
On Min Value	The Unity Event invoked when the current value reaches the minimum value.
On Max Value	The Unity Event invoked when the current value reaches the maximum value.

On Value Changed	The Unity Event invoked when the current value has been changed. This is invoked with the change in value and the direction of damage, if using Health.
On Load Condition True	If Save is enabled, the Unity Event invoked if the restored value is greater than zero (True) on Start.
On Load Condition False	If Save is enabled, the Unity Event invoked if the restored value is zero (False) on Start.

Method	
GetValue()	Returns the current value.
Save()	Save the current value.
SetValueAndSave(float value)	Set the current value and save it.
Increment(float value)	Increment the current value.
SetValue(float value)	Set the current value.
CanIncrement(bool value)	If false, the current value can't be incremented.
SetTrue()	Give the current value a value of 1f.
SetFalse()	Give the current value a value of 0f.
CanTakeDamage(bool value)	If this is Health, the layer on this gameobject will be switched to the ignore raycast layer if False. Will revert on True.

World Float HUD

WorldFloatHUD is a convenient class for displaying world float values using UI elements. This class can also work with other classes like Projectile and Firearms to display ammunition values. The UI element style is left to your personal taste. This class will simply enable images and set values.

Property	
Type	Discrete items will display the value in terms of UI images. Think health hearts. Specify the sprites that represent an empty and full value. Must also set number of UI images that will be used to display the value. Continuous will display the value by modifying the fillAmount of an image. Numbers will use a TextMesh to display the value.
Value Type	Only set the reference for one of these to display the value. For Projectile, Firearms, and Tool the value represents ammunition amount. To set Firearms simply drag the player gameobject into this field. This will display the ammunition amount for the current firearm the player has active. To set tool, drop any Firearm into this field.

String And Vector3

These work in a similar fashion as Float. The respective scriptable objects are **WorldStringSO** and **WorldVector3SO**.

Events	
AfterLoad	If Save is enabled, the event invoked after the value has been restored. This occurs during Start.

Method	
GetValue()	Returns the current value.
Save()	Save the current value.
SetValueAndSave(type value)	Set the current value and save it.
SetValue(type value)	Set the current value.