ARCADE IDLE ENGINE User Manual

Quickstart

> **WARNING**
>
> Please import DOTween before the asset. It’s free!

Up-to-date documentation: https://www.arcade-bridge.com/arcadeidleengine-docs/

If you are encountering problems, please come to our Discord server. If you don't have a discord account, you can send an e-mail: `thearcadebridge@gmail.com`

If you are creating a new project, then make sure to switch the build target to mobile (Android/iOS) and then import the DOTween. Asset contains Odin Serializer which requires the project to be in `.Net Framework` instead of `.Net Standard`.

After importing the asset, you will see two scenes. **Showcase** scene is all about showing all the features in the most straightforward way while **Sample Game** scene tries to replicate a typical Arcade Idle game scenario. You can find both of the scenes under the directory: `ArcadeIdleEngine/Demo/_Scenes/`

In the next section, we’ll explain systems, so you have better understand overall. If you are still confused, analyze scenes and prefabs to understand how they work. I did my best to simplify as much as possible but because it's an Arcade Idle genre, it gets quite complicated pretty easily.

> Thanks for buying the asset! If you liked it, please head to the store page for review.

Concepts

These explanations offer comprehensive insights into the underlying concepts. The systems are intentionally crafted with the assumption that documentation may not be necessary, but this space is available if you find any difficulty in grasping the concepts.

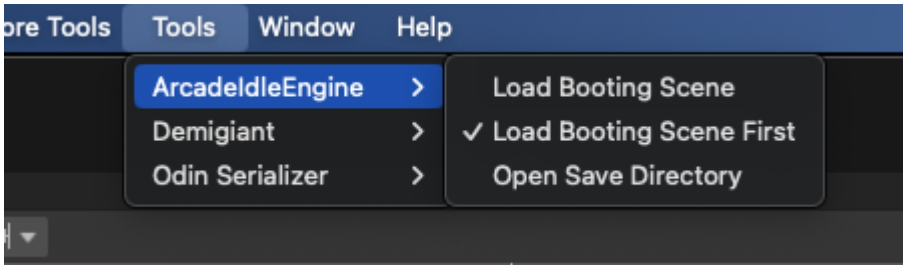
All systems are designed with simplicity, easy extensibility, and maximum isolation in mind, enabling independent use. The codebase has been meticulously constructed for clarity, ensuring that no component engages in unnecessarily complex calculations or enigmatic actions.

Actors

The responsibility of moving characters, primarily the player, though other characters like workers can also utilize it, lies within this system. Components such as `HumanoidAnimationManager` are encompassed here. Additionally, it houses the `InputChannel` for facilitating communication between the UI (Joystick) and the game world (player character).

Booting

The Booting scene contains a collection of GameObjects responsible for initializing the game. It loads save data and potentially runs other types' initializations, such as SDK initializations or fetching remote config data. `GameBooter` waits for all this and then loads a level. When you initiate playback in the Editor (and during runtime), if `Load Booting Scene First` menu item is enabled, then it will load booting scene first and then loads the level you were currently in.



Data

Contains data types and tools to let you store data like score, coin, collected resource count and so on. They can be stored using any `Saveable` class.

Save System

Arcade Idle Engine contains a lot of ScriptableObjects, some of them is just plain data holder but others have functionality as well. ScriptableObjects derived from Saveable

contain implementation for saving data. These data can be saved and loaded from the disk. Initial value can be set if there isn't any save file. These `Saveable` objects can be saved by `SaveManager` class.

You need to assign Saveables to the `SaveManager` so it can save and load them when the game runs. Use SaveManager to save your game state like collected wood count or money. Use `Tools > HypercasualPack > Open Save Directory` if you want to delete the save file or modify it manually.

> **TIP**
>
> When working in Unity Editor, if you enabled `Load Booting Scene first`you might want to disable the `Save Automatically` bool in the `Automatic Save Load` GameObject inside the `Booting` scene in order to test things without actually saving them. Because otherwise, `Automatic Save Load` object will persist between scenes and saves everything when you exit playmode.

Saving tool list is more complex because it's needed to save their indexes. `UniqueIntListVariable` can be used to save their indexes. And then `GatheringToolDefinitionIndexLookup` can be used along with `GatheringToolDatabase` for indexing every gathering tool.

You can also extend classes to create your own implementation by creating your own custom implementation that derives from `ObjectDatabase` and `IndexLookup` or `Saveable<T>`. Two concrete implementations can be found in the asset: `Int Variable` and `UniqueIntListVariable`.

> **TIP**
>
> There is also a `SaveUpgrader` class that handles new save versions. It's usually required when you change the save data. If you need to learn more about this save versioning concept, you can find a lot of explanations on the internet.

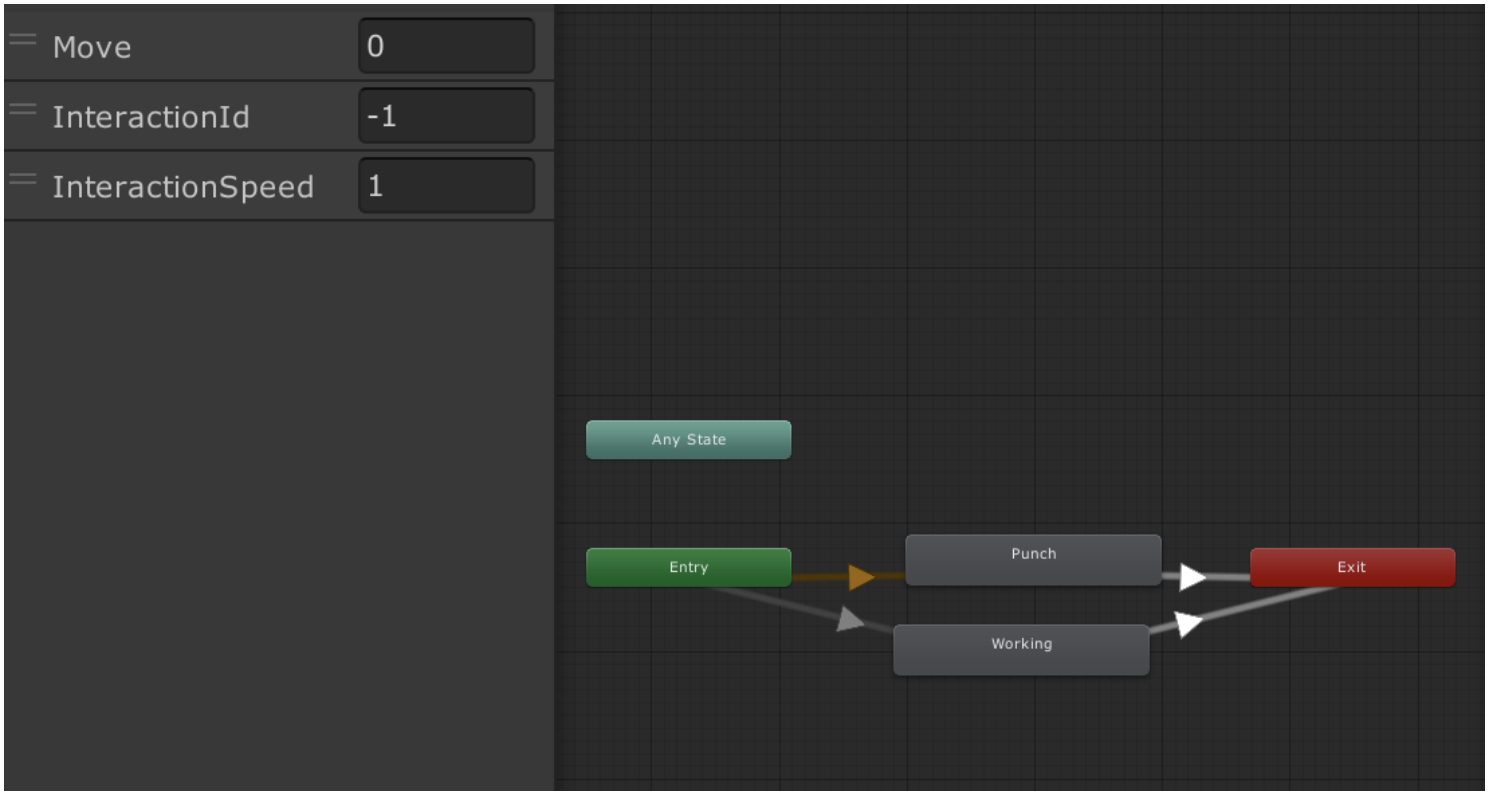
Economy

Utilizes Data system to spend resources (coin, gem, wood). It can animate earning income via `ResourceAnimators`.

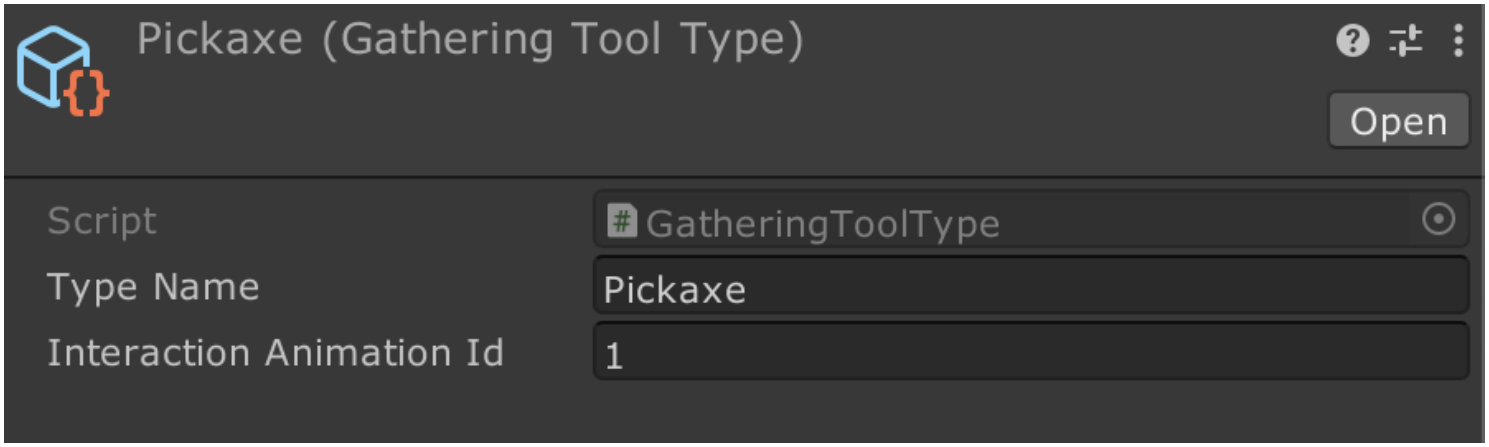
Gathering

Contains tools and gatherable sources which can be anything from trees to ores. You can get creative and invent interesting gathering mechanisms. Because essentially this system lets player produce an item if he has the right tools to gather. So one original idea might be a coffeemaker as a gathering tool and a sack full of coffee bean as a gatherable source. You grind and brew them and you get coffee cups.

For setting custom animations, you can create new state in the `Actor` animator and create a new transition from Entry to Exit with an equality check.



Then assign the `InteractionAnimationId` in the `GatheringToolType`.



Interactables

Contains all the objects that can be interactable in the game world. It can be pop-up displayer, or a trigger that collects items from inventory when an inventory manager enters

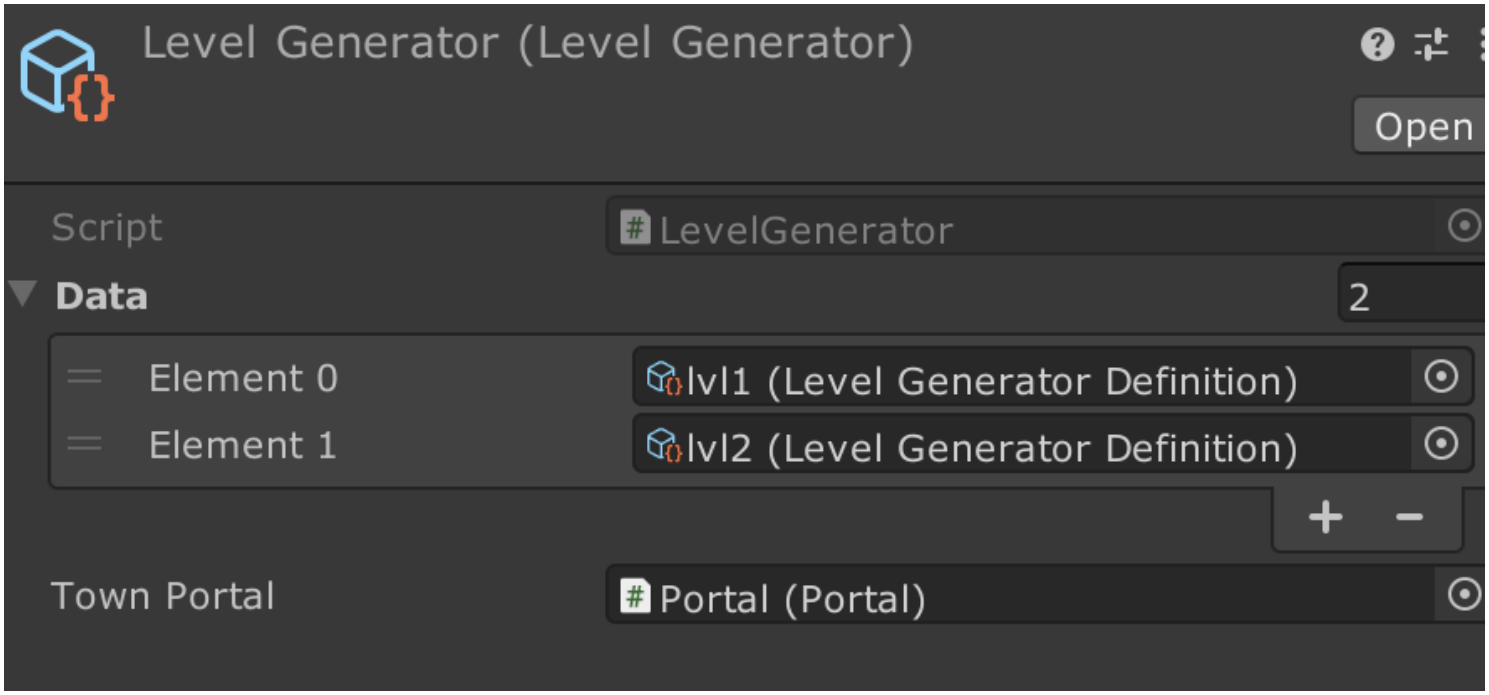
the area.

Inventory

Nearly every system relies on the inventory. Spawner spawns items into the inventory, Transformer gets item from one and produces an output to the other. Systems like `InventoryFeederTriggerArea` and `InventoryCollectorTriggerArea` directly interact with the `Inventory` to add, remove, or get the desired Items.

Level Generation

The Level Generation system comprises components such as `LevelGenerator`, `LevelGeneratorDefinition`, `Group`, `Tile`, and `Portal`. Additionally, the `LevelGenerationTrigger` is included for initiating new level generation, provided as an example for customization. It's quite straightforward.



Each entry in the LevelGenerator represents a distinct world. You can have as many different levels as needed. When generating a new level, specify the desired level to create. In the example above, there are 2 levels.

Each element in LevelGeneratorDefinition corresponds to the difficulty level. So for example:

```
Element 0 = difficulty 0
Elemnt 1 = difficulty 1
```

If there's more than one entry inside the Group in the Element, a random one within that Group will be chosen. You can also specify a Group for the end of the level (e.g., a boss), and which tile to use when spawning all the groups.

Monitors

UI Monitors

These are basic objects that visually represent data, usually through a text object within a Canvas. They actively listen for changes in the variables they are monitoring and automatically update their displayed values accordingly.

Timer Monitors

Displays time information to the player via images. It's so easy to create your own monitor that presents information in a different way.

Count Monitors

Displays count information. It is used with the inventory to show what item it contains and how many.

Items

If something can be picked up, it can be turned into a `Item`. To mark an object as Item, just add a Item component to it; think of it like a tag.

There's a significant part called `ItemDefinition`, which includes details like whether the picked item should be visible, if it can be sold, and its image for the UI. Many other systems need this Item type.

Pools

These are generic implementations of pools. It's crucial to use them instead of frequently using Instantiate and Destroy because doing so would put unnecessary strain on the CPU and memory, resulting in additional garbage in the memory.

Processors

These are kind of like a machines which processes Items. Process can do selling, spawning, or transforming.

Sellers

It sells desired Items by using their sell value and sellable properties. There are 2 kinds of Sellers. One is `ItemSellerFloatingText` and the other is `ItemSellerFloatingImage`

Some `IntVariable` are treated with extra stuff and they are called `Resource`. These are things like coin, money, wood or so on. You can use `ResourceAnimator` along with the `ResourceTargetImage` to create animated image effect when resource changes. You also need to make sure that the GameObject has the IntVariableMonitor.

Spawners

`ItemSourceSpawner` spawns Items using a pool. It can be then collected by a GameObject that has `InventoryManager` component.

Transformers

The script defines a class that is responsible for collecting, modifying, and stockpiling items according to specified rules and timers. This class manages the transition of items between unmodified and modified states, with the modification process influenced by an upgradeable work speed. It utilizes coroutines and timers to manage item collection and processing.

> **WARNING**
>
> Ensure that the number of text fields in `ItemTransformerMultipleCondition` matches the number of input fields in the `MultipleConditionRuleset`. If they do not match, an error will be printed out in the console.

Workers

Workers can be spawned using `WorkerManager`. You can controll all the workers who has been spawned from the same manager. In the asset, you can see two most common features, pausing and resuming all the workers.

There are two different versions of workers, one has the extra `ItemGatherer` so they can gather gatherables. Create new worker prefabs and assign them to the WorkerManager's so you can spawn and control them.

Tween Feedbacks

Simple scripts that allows you to define any custom feedback effects. You can create new feedback effects by changing the properties, or implementing new behaviours by deriving from `TweenFeedback`. It's mostly used in gathering, when chopping trees for example.