Documentation – FREE

JUNE 2022

Dragon Arts s.r.o.

Introduction: https://dragonarts.sk/projects/rewarder

Documentation: https://dragonarts.sk/docs/rewarder

Support: support@dragonarts.sk

RED Parts are not applicable for this LIMITED Version.

Dependencies

- TextMeshPro
- DOTween

NOTE: *Applies to examples only*

Installation

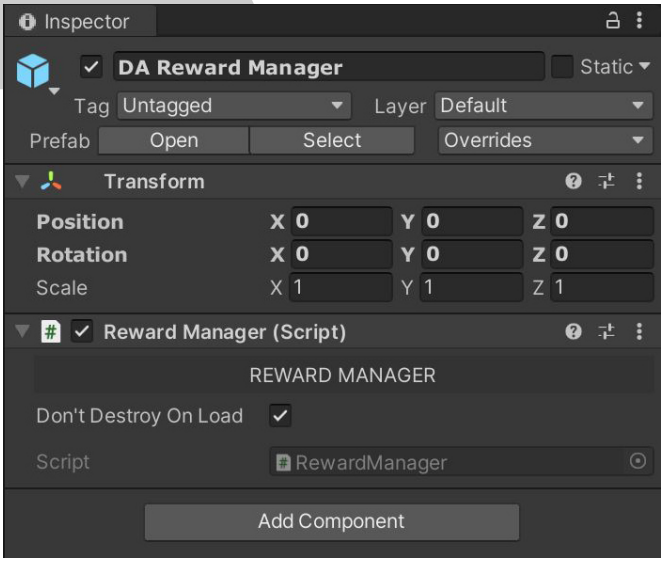
Just import all files into your project.

Usage

Reward Manager

In order to make use of Rewarder's functionalities, you have to establish a Reward Manager:

Hierarchy Window (Right Click) > Create Other > Dragon Arts > Reward Manager



NOTE: *It is recommended to create the Reward Manager in your persistent scene.*

Reward Manager represents a single source of truth, which controls Drop Tables & Timers.

After the initiation in your scene, you can simply use its reference and call its methods anywhere in your code, e.g.:

```
RewardManager.Self.PickRewards(dropTable, 1);
```

NOTE: *Do not forget to import namespace:*

```
using DragonArts.Rewarder;
```

There are only few methods to operate with:

- Pick exact number of rewards from Drop Table

```
PickRewards (DropTable dropTable, int optionalCount, PickOptions options = null)
returns PickResult
```

- Pick random number of rewards from Drop Table

```
PickRewards (DropTable dropTable, int[] optionalRange, PickOptions options = null)
returns PickResult
```

- Register the Timer

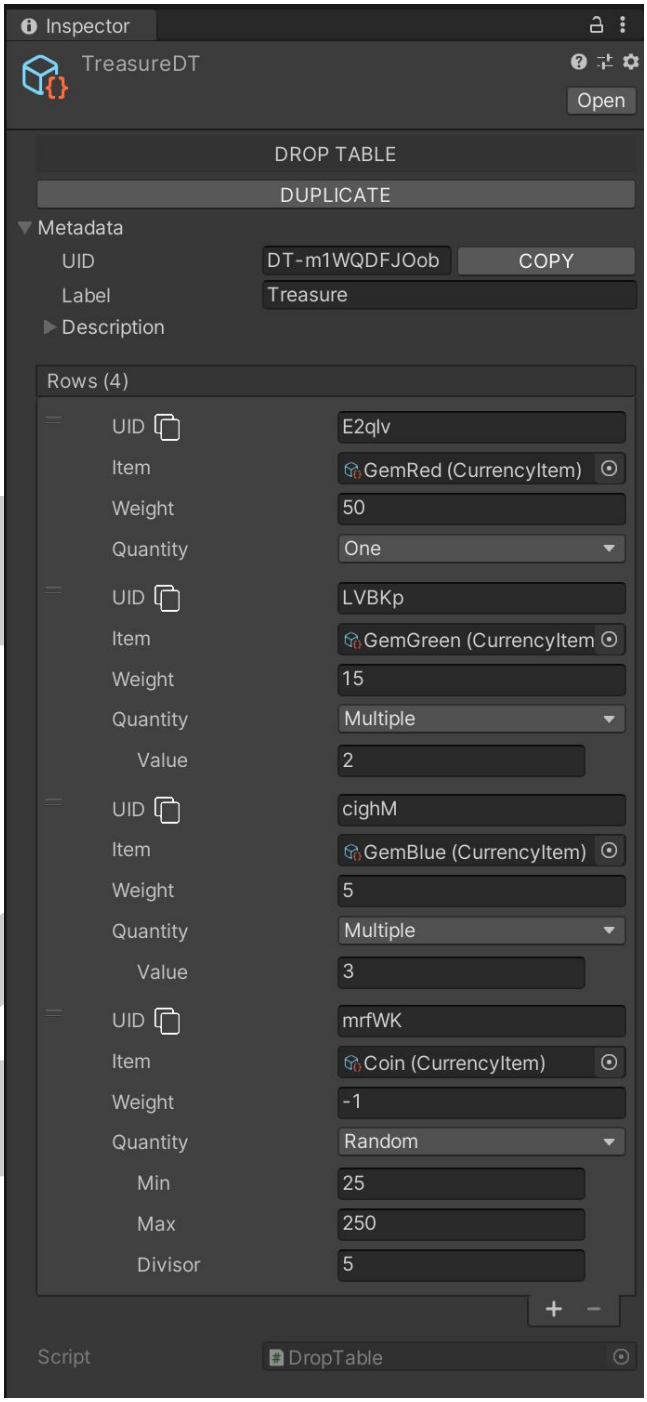
```
RegisterTimer (Timer timer, TimerOptions options)
returns TimerInstance
```

- Getters & Setters

```
GetGlobalRandom ()
returns System.Random

SetGlobalRandom (int seed)

GetGlobalTime ()
returns System.DateTime

SetGlobalTime (System.DateTime time)

GetGameTime ()
returns System.DateTime

SetGameTime (System.DateTime gameTime)
```

Drop Tables

NOTE: *Before you continue, you should read the documentation about Scriptable Items & Groups*



DropTable is a derived class from class ScriptableItem just like ScriptableGroup, so it inherits metadata attributes (UID, Label & Description).

Besides that, each Drop Table has its own Reorderable List of classes DropTableRow, which represents the rows of the table.

Each row has its own UID, Item, Weight and Quantity.

UIDs can be easily copied to clipboard by clicking on

Item represents the Reward. It can be any derived class of ScriptableItem, which means that it can be also a ScriptableGroup or even another DropTable.

Weight is the main variable use at picking rewards. The value of -1 means, that it will be picked no matter what. Except of that, the higher the number the greater the chance of picking. 0 means, that it will be not picked.

Rewards can be also quantified. There are 3 options:

- One – Drops only 1 Item
- Multiple – Drops the exact number of Items
- Random – Drops random number of Items between defined range. This number of items will be divisible by defined divisor without remainder.

Project Window (Right Click) > Create > Dragon Arts > Drop Table

Rewards Picking

Picking the rewards is really straightforward, you just need to call the **PickRewards** method of the Reward Manager and provide the source Drop Table, from which you want to pick rewards, e.g.:

```
RewardManager.Self.PickRewards(dropTable, 3);
```

Let's asume, that this dropTable has 1 row with weight of -1 and 5 rows with weight greater then 0. In this case will Reward Manager pick 4 Rewards. It will be that row with weight of -1 and 3 more rows from other 5.

There is also a possibility to pick random number of rewards:

```
RewardManager.Self.PickRewards(dropTable, new int[] { 2, 5 });
```

In this case will Reward Manager pick guaranteed rewards (rows with weight of -1) and additional 2, 3, 4 or 5 other rewards.

NOTE: *The random number of rewards or even the random quantity are computed by global System.Random instance, which you can freely set before picking by* ***SetGlobalRandom*** *method of the Reward Manager.*

If you want to pick only guaranteed rewards, you can do that also:

```
RewardManager.Self.PickRewards(dropTable, 0);
```

If you need more control over rewards picking process, you are free to provide PickOptions as the third parameter of this method.

Pick Result

The output of method PickRewards is an instance of class PickResult. PickResult contains an updated instance of PickOptions and a list of instances of class **Reward**.

Reward is a wrapper class, which contains the picked Scriptable Item, its quantity and the paths that were taken during the picking process. The path consists of UIDs of visited rows.

You can get Reward's Path or UID easily by calling its method **GetPath** or **GetUid**. In the case of Reward instances, the UID is a Path joined by dash.

Pick Options

By providing the PickOptions as the third parameter of method PickRewards, you are able to override default picking behavior:

```
RewardManager.Self.PickRewards(dropTable, 2, new PickOptions {
      seed = seed,
      snapshot = snapshot,
      replacements = new List<PickReplacement>() {
            new PickReplacement () {
                  uids = uids,
                  blacklist = false,
                  bulk = false,
                  replace = (int weight) => {
                        return weight - 2;
                  }
            }
            ...
      },
      conditions = new List<PickCondition>() {
            new PickCondition () {
                  uids = uids,
                  blacklist = false,
                  startup = false,
                  validate = () => {
                        return false;
                  }
            },
            ...
      },
      modifiers = new List<PickModifier>() {
            new PickModifier () {
                  uids = uids,
                  blacklist = false,
                  type = PickModifierType.WEIGHT,
                  modify = (int[] rowValue) => {
                        rowValue[0] *= 2;
                        return rowValue;
                  }
            },
            ...
      },
      group = false
});
```

Seed

The value of attribute Seed is changing at every execution of method PickRewards by default. By providing the Seed value, you are able to control the randomness of picking and even repeat randomness.

Snapshot

Snapshot is a flat representation of Drop Table in time. It is a simple list of classes DropTableSnapshot. By providing it in PickOptions you are able to override Drop Table's default weights or quantities before picking. You are free to serialize it e.g., into JSON and store, so you will be able to recover the last state of Drop Table next time you will need it.

NOTE: *Snapshot is frequently used in combination with Replacements.*

Replacements

There could be cases where you will need to pick rewards without replacement, which means that when is a Reward picked, its weight is decreased, so next time there is lower possibility to be picked. When it reaches the weight of 0, then it will be not picked anymore.

There are multiple possibilities to define affected Rewards:

- By providing the UID of Drop Table
- By providing the UID of Reward
- By setting the value of attribute Blacklist
  - True means, that PickReplacement will affect any Reward except of those matched with UID
- By setting the value of attribute Bulk
  - True means, that when is picked a Reward, which matches the above criteria, then the PickReplacement will immediately affect too each other Rewards, which matches the criteria.

You are free to implement the **Replace** function of PickReplacement instance. If there is no function defined then the default routine will run, which is decreasing the weight by 1.

NOTE: *Define as much PickReplacement instances as you want, just keep in mind, that they are executed in defined order. Same applies for Conditions and Modifiers too.*

Conditions

Conditions excludes Rewards from PickResult, which matches the criteria. Like at Replacements there are multiple possibilities to define affected Rewards:

- By providing the UID of Drop Table
- By providing the UID of Reward
- By providing the UID of ScriptableItem
- By setting the value of attribute Blacklist
    - True means, that PickCondition will affect any Reward except of those matched with UID

NOTE: *By providing the UID of ScriptableItem, you are excluding each row with that item i.e., each Reward.*

By setting the value of attribute Startup, you are able to decide whether you want to exclude matched Rewards before picking or after picking straight before building PickResult instance.

Implementing of the **Validate** function of PickCondition instance is required.

Modifiers

Modifiers are used to modify the Weight or Quantity of Rewards, which matches the criteria. The difference between Modifiers and Replacements is

that, that the **PickModifier's modifications are not updating the Snapshot**, while at Replacements are modified weights stored in Snapshot. Also, Modifiers are able to modify weights even before picking.

Like at above 2 there are multiple possibilities to define affected Rewards:

- By providing the UID of Drop Table
- By providing the UID of Reward
- By setting the value of attribute Blacklist
  - True means, that PickModifier will affect any Reward except of those matched with UID

By setting the value of attribute Type you are deciding whether you are going to modify the weight or the quantity of Rewards.

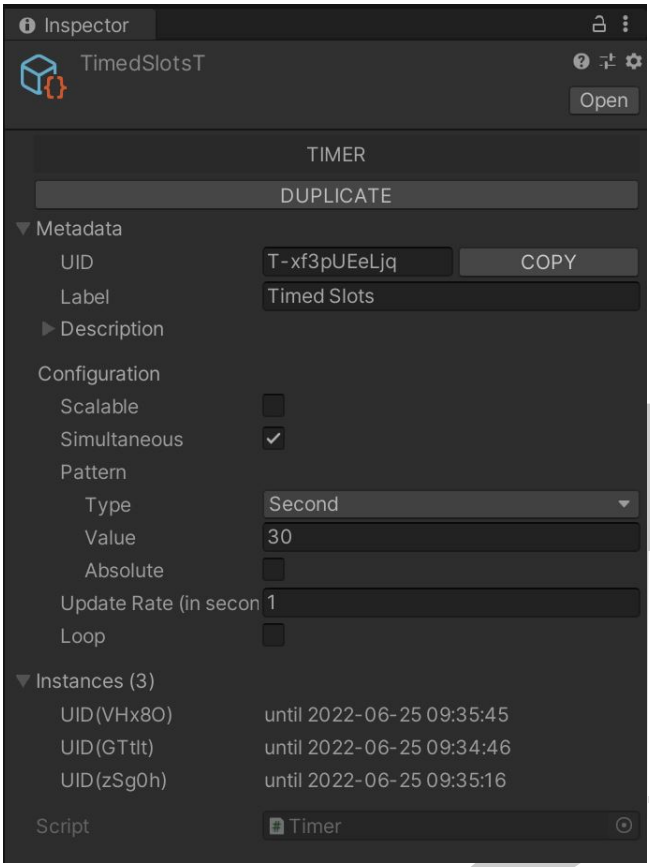
Implementing of the **Modify** function of PickModifier instance is required.

NOTE: *The parameter of function Modify is an array of integers. This is because at Random Quantity you have to modify a range.*

Group

The value of attribute Group is telling whether you want group Rewards in PickResult based on ScriptableItem UID or not. Grouped Rewards may have multiple Paths and so multiple UIDs. By this, you will always have a chance to find out from where those Rewards came from.

Timers



Timer is also a derived class from class ScriptableItem, so it inherits metadata attributes (UID, Label & Description).

Besides that, each Timer has its own configuration and a list of active instances.

Global vs Game Time

Rewarder define 2 types of Time. First is a Global Time, which is updating in Coroutine every 1 second. This Time is **not affected** by **Time.timeScale**. The second type is a Game Time, which is updating every few milliseconds through MonoBehaviour's Update method, so it is **affected** by **Time.timeScale**.

Project Window (Right Click) > Create > Dragon Arts > Timer

NOTE: *There is a possibility to do a time leap by setting the time manually by **SetGlobalTime** or **SetGameTime** methods of the Reward Manager, which could come very handy during debugging.*

Timer Registering

You are free to register any number of Timers, through the **RegisterTimer** method of the Reward Manager, you just need to provide a source Timer and optionally an implementation of onComplete and onUpdate routines:

```
RewardManager.Self.RegisterTimer(timer, new TimerOptions {
      onComplete = () => {},
      onUpdate = (System.TimeSpan timeLeft) => {}
});
```

By registering a Timer, you are creating an instance of that Timer, a TimerInstance, which makes use of the Timer's configuration (TimerConfiguration) and the provided options (TimerOptions).

Timer Configuration

Scalable

By setting the value of attribute Scalable you are deciding whether the Timer should use Global or Game Time. True means, that it will use Game Time.

Simultaneous

Checking the Simultaneous checkbox means, that there could run more active instances of this Timer simultaneously. By default, is allowed only 1 active instance of the same Timer.

Pattern

The Pattern is the most important attribute, which is used to calculate the end of the Timer. By pattern you are basically defining the duration of the Timer, which can be in milliseconds, seconds, minutes, hours, days, months or years. There is also a possibility to define an Absolute Pattern by checking the Absolute checkbox. Let's say, we have just registered a Timer with an Absolute 1 Day Pattern. The Timer will end at the start of the next day. In contrast with a Relative 1 Day Pattern, which would end after 24 hours.

Update Rate

The Update Rate controls, in what intervals should the Reward Manager call the onUpdate routine.

Loop

Lastly, the value of attribute Loop is telling whether to restart the Timer after the onComplete routine is called or no.

Timer Options

As you can see above, by the instance of TimerOptions you are able to implement the onComplete and onUpdate routines, but there is more:

```
RewardManager.Self.RegisterTimer(timer, new TimerOptions {
      onComplete = () => {},
      onUpdate = (System.TimeSpan timeLeft) => {},
      startTime = System.DateTime.Now.AddYears(1),
      endTime = System.DateTime.Now.AddYears(2),
      pattern = new TimerPattern {
            type = TimerPatternType.Day,
            value = 2,
            absolute = true
      },
      updateRate = 2,
      loop = true
});
```

There is a possibility to define the Start and End Time and also override the Pattern, Update Rate or Loop attribute from TimerConfiguration.

Start Time

In case that the attribute StartTime is not provided, then System.DateTime.Now is used.

End Time

The EndTime takes precedence before the Pattern. If it is provided, then the Pattern is ignored. The Pattern will be used again the next EndTime calculation e.g., at Timer's restart.