ARES

– Turn-based Battle Engine –

V1.3

Introduction

Philosophy

ARES is a turn-based battle engine designed with flexibility and ease of use in mind. Being well aware that each game is different and will have its own unique requirements, ARES aims to provide a solid base for game developers to build upon, rather than trying to cater to every possible use-case out of the box.

This may sound a bit unintuitive, but there are several advantages to this approach:

- Designers don't have to trawl through countless options and settings that their game will never use.
- Developers don't have worry about breaking obscure parts of the code-base when extending or modifying the engine.
- The code base remains cleaner and more concise, allowing for far easier maintenance.
- There is less boilerplate and unused code to run at runtime, resulting in more optimized builds.

Of course I am very aware that modifying or extending unknown code can be a daunting task at best. As such, an extreme amount of care has been given to make this process as easy and painless as possible, and some common examples will been given in this document.

In addition to making the engine easy to extend, a large number of hooks have been provided in the form of *UnityEvents*, allowing developers to listen and react to events from external scripts. If possible, this workflow is recommended for most cases as it foregoes the need to understand and modify Ares' internal workings.

The end result is a flexible system that can be extended, broken apart and rebuilt with relative ease; either by building a layer on top of ARES or by modifying its internal workings.

Suitable gametypes

One of the most common reservations when it comes to considering assets like this are the questions of *"will this asset do what I need it to?"* and *"will this asset work well with my unique game type or architecture?".* The answer, as expected, is unfortunately a bit hard to put on paper like this.

Probably the most obvious game types that come to mind are turn-based RPGs like those from the Final Fantasy or Pokémon series, and indeed ARES lends itself extremely well to these kinds of setups, and has been designed with games like these in mind.

However, even games that don't follow this formula can make great use of the engine by building systems on top of it, or running them on the side. A less obvious example might be Tactical RPGs like Fire Emblem, where ARES could either be extended to have a "move" and "attack" phase, or it could run alongside your movement code and only handle the battle logic.

Even real-time battle games might find some use in ARES, as the functionality handling abilities, items, afflictions and the like could just as easily be used in non-turn-based environments.

Depending on your project, obtaining a solid base for all of these can save a lot of time and frustration, with things like effect chaining and relations between stages of ability already being taken care of (in the form of action chains; for more information see Actions and Action Chains).

Whatever your game is like, if it handles combat in some way – especially turn-based – ARES is likely to be a good fit for it.

If you have any lingering questions or aren't quite sure if ARES is for you however, feel free to send an e-mail to p_boelens@msn.com and I will try my best to answer any questions you may have.

Table of Contents

System Design

Introduction

Every battle starts with the creation of a **Battle** object through code. This object handles the main flow of the battle and keeps the state. Battles consist of rounds, which are further divided into turns.

The fight itself is fought by **Actors**. These can be either player-controlled or be run by some AI. Within a battle, each actor is assigned to one of multiple **Battle Groups**. At least two such groups are required per battle.

Actors spend their turns either casting **Abilities** or using **Items**. They can also become afflicted by **Afflictions**; status conditions that can have various effects, good or bad.

Lastly, the battle can also be affected by **Environment Variables**. These can be used to create literal environment changes like weather effects, or more general "aura" effects.

With all this going on there are a lot of settings involved, like in which order to trigger certain effects, or what to do when there are no valid abilities for an actor to cast. All of this can be set in a **Battle Rules** object. Each **Battle** object requires one of these to determine the flow of battle.

Editor Preferences

Ares has a couple of editor UI features designed to make your development life easier. That said, they might not always be wanted, as they could also hinder you in some cases. As such, some editor preferences have been made available in the Preferences window (`Edit > Preferences` on Windows, `Unity > Preferences` on MacOS).

Editor Preferences	
Show ability options on list add Show affliction options on list add Show item options on list add Show item group options on list add	For each of their respective objects, clicking the **+** button on a reordorable list containing them will show a dropdown menu with all available options. If the preference is disabled it follows the default behaviour instead, simply adding a new entry with it's settings duplicated from the prior one.
Show instantiation effect as enum	Similar to the above, but for `AbilityEffect` entries in instantiation effects. Rather than showing a raw object field, it scans the `Assets` folder for any valid effects and displays those as an enum field instead.
Use verbose debug logging	Verbose debug logging enables a group of internal logs. These are particularly useful when extending the engine, or when you need a clear grasp on which events are fired when, and how and when internal state changes. With the setting disabled, these internal calls are stripped from the code entirely when you build your game.

Regular verbose logging color State 1 verbose logging color State 2 verbose logging color Action verbose logging color Unimportant verbose logging color	Only available when verbose debug logging is enabled. These colors reference the 5 default colors used for verbose logging. This helps differentiate between the different types of messages and makes the log easier to scan.

Namespaces

All ARES classes and functionality live inside the `Ares` namespace. Most classes you will interact with will be in this namespace directly, but there are a few nested ones as well.

Ares namespaces	
Ares	Contains most classes, including those you are most likely to want to hook into. Classes like `Battle`, `Actor` and `Ability` live here.
Ares.ActorComponents	Contains the `ActorAnimation`, `ActorAudio`, and `ActorInstantiation` classes, as well as all of their helper and wrapper classes.
Ares.Development	Contains the `Readme` class used in the example scene, as well as the menu entries for the Unity editor and a generic template for a custom battle manager script.
Ares.Editor	Contains all `CustomEditor` and `PropertyDrawer` classes.
Ares.Examples	Contains all classes created specifically for the example scene, housing things like the battle manager, custom callback methods and scripts for controlling the environment.
Ares.Extensions	Contains the `ExtensionMethods` class which houses a few extension methods for lists and Unity Events.
Ares.UI	Contains all classes relating to UI elements, like the `HPBar`.

Actors

Every **Actor** requires an `Actor` component. ARES ships with two by default: a **Player Actor** and a very simple **AI Actor**. These both derive from the `Actor` class, and can be overridden or extended as desired. Player Actors wait for input via callbacks whereas AI actors get their decision-making methods called directly. For more on these callbacks, see the Battle Events.

The Actor component has a couple of exposed properties that require some further explanation.

First, the various stat values are at base level - meaning they are the values to which buffs and debuffs get applied. The stats themselves are taken from the **Stat** objects that exist in the `Resources` folder (see the Stats documentation below for further information on this).

Second is the `Abilities` list. This is where you can define which actor has which abilities. One way of using this list is to populate it only with abilities the actor can use. Another approach could be to populate it with all the abilities an actor *could* have, and only enable those it does. This might be more desirable for prefab-based workflows.

The base duration of each ability can be overridden in case a particular actor needs more or less time than the default.

Lastly, the fallback ability can be used to provide an ability that gets fired when all other abilities are unavailable during a battle, or when the selected action is no longer valid. This behaviour depends on the settings chosen in the **Battle Rules**.

Aside from these properties, a few events and some internal logic like hit testing, the `Actor` class itself is relatively barebones, mostly handling state and communicating with the `Battle` class.

Instead, different components are used to trigger or override animations, sounds and instantiations. This decoupling allows for more maintainable code, and allows you to only use the components that you actually need.

The `Actor` component warns you about any potentially missing components in its Inspector, and all default components can be added at once from Unity's Component menu located in the header bar (`Component > Ares > [AI | Player] Actor + Actor Components`).

Events

Besides these properties, actors fire a number of events that you can listen to and hook your own custom logic into. These are:

Actor events	
OnHPChange *int newHP*	Invoked when the actor's HP changes. Compare this to the actor's current HP (`actor.HP`) to determine the amount with which it will change.
OnHPDeplete	Invoked when the actor's HP hits 0.

The Actor component also fires several events related to Abilities, Items, Afflictions, Stats and player input. These are described in full in their respective sections.

Actor Animation Component

The **Actor Animation** component handles the playing of Mecanim animations on actors. It does this by listening to relevant events and setting animator parameters accordingly. These parameters can then be used to drive transitions within your Animator Controller. All parameter types are supported.

Parameters can be reset immediately after setting by the component, or they can be left for your Animator Controller or custom scripts to deal with.

ARES is not aware of and does not care about what your animator does with its parameters, and can thus be used with both 2D and 3D animations.

The Actor Animation component is able to request battle delays to ensure all animations are completed before continuing the battle. It does this by adding a `BattleDelayElement` component in its `Start()` method. (For more information on how this works, see Timing and Battle Delays below.)

There are three delay options:

Actor Animation delay options	
None	Doesn't request a lock or delay.
Animation	Requests a delay lock for the duration of the animation. Because ARES does not know which animation will play – and it might be dependent on external factors with more complicated Animator Controllers – the following method is used instead: 1. Monitor the set variable and wait until its value changes. This indicates that the value has been consumed. 2. Wait until the controller is no longer in any transition. 3. Wait until either the current active animator state on the specified layer changes, or until that state's `normalizedTime` is 1 or greater. While not completely fool-proof, this approach handles most scenarios, including switching between states from multiple different states and transitions, and final animations that prevent the controller from evaluating further (i.e.: a final "defeat" animation).
Time	Requests a timed delay as specified per action.

The battle delay reason used is always `Animation`.

Actor Audio Component
The **Actor Audio** component handles the playing of sound effects. If a sound effect is attached to the used item, ability, affliction or environment variable this component becomes responsible for playing it.

In addition to the audio clips attached to these actions, the actor might want to play its own clip in response to it as well. Callbacks for this can be added on this component as well.

Actor Instantiation Component
The **Actor Instantiation** component handles the creation of instantiation effects. If an instantiation effect is attached to the used item, ability, affliction or environment variable this component becomes responsible for spawning it.

In addition to the instantiations attached to these actions, the actor might want to spawn its own objects in response to it as well. Callbacks for this can be added on this component as well.

Battles

Overview
Battles are responsible for the main state and flow of the battle. They keep track of the actors and related information, progress and cycle between different states and handle all of the effects.

Each will battle will follow the rules as laid out inside a **BattleRules** object. These rules dictate things like the action select moment, action fallbacks, which kind of delays are allowed and more. For more on this, see Battle Rules.

Participation
When sorting, queuing and processing actors and their effects, only participating actors are taken into consideration.

Non-participating actors can still be targeted by certain items or abilities if they are set up to. Actor participation can be set from script by calling the `Battle.SetParticipation()` method.

Progression

By default the battle will continue automatically whenever it can, but it is possible to disable this behaviour and manage this progression yourself. This option is found in the Battle Rules object. Note however that this leaves the burden of managing various timings and delays on your custom handler script.

Creating a battle

Battles are created and set up entirely through code. This typically happens from some kind of **Battle Manager** script. As each game is different and will have unique requirements it is up to individual developers to manage battles' lifecycles, UI and events.

The demo scene comes with an example `BattleManager` that handles all of these things. In practice, depending on your setup, you might want different scripts for handling these different responsibilities.

A stub is provided in `Ares/Scripts/Development/BattleManagerTemplate`. This can serve as a starting point for your own manager script, outlining which steps typically need to be taken care of.

Ending a battle

Different battles might have different win conditions, or there might be special battle events that are hard to account for. As such, determining the winner is left entirely to outside scripts.

For a lot of typical win conditions, one could listen to the `BattleGroup.OnDefeat` event. From there you could determine if the group is friendly or not, if defeating this one group is enough to call the match, etc. If it is, you will then need to call your battle's `EndBattle()` method with the `Battle.EndReason.WinLoseConditionMet` reason.

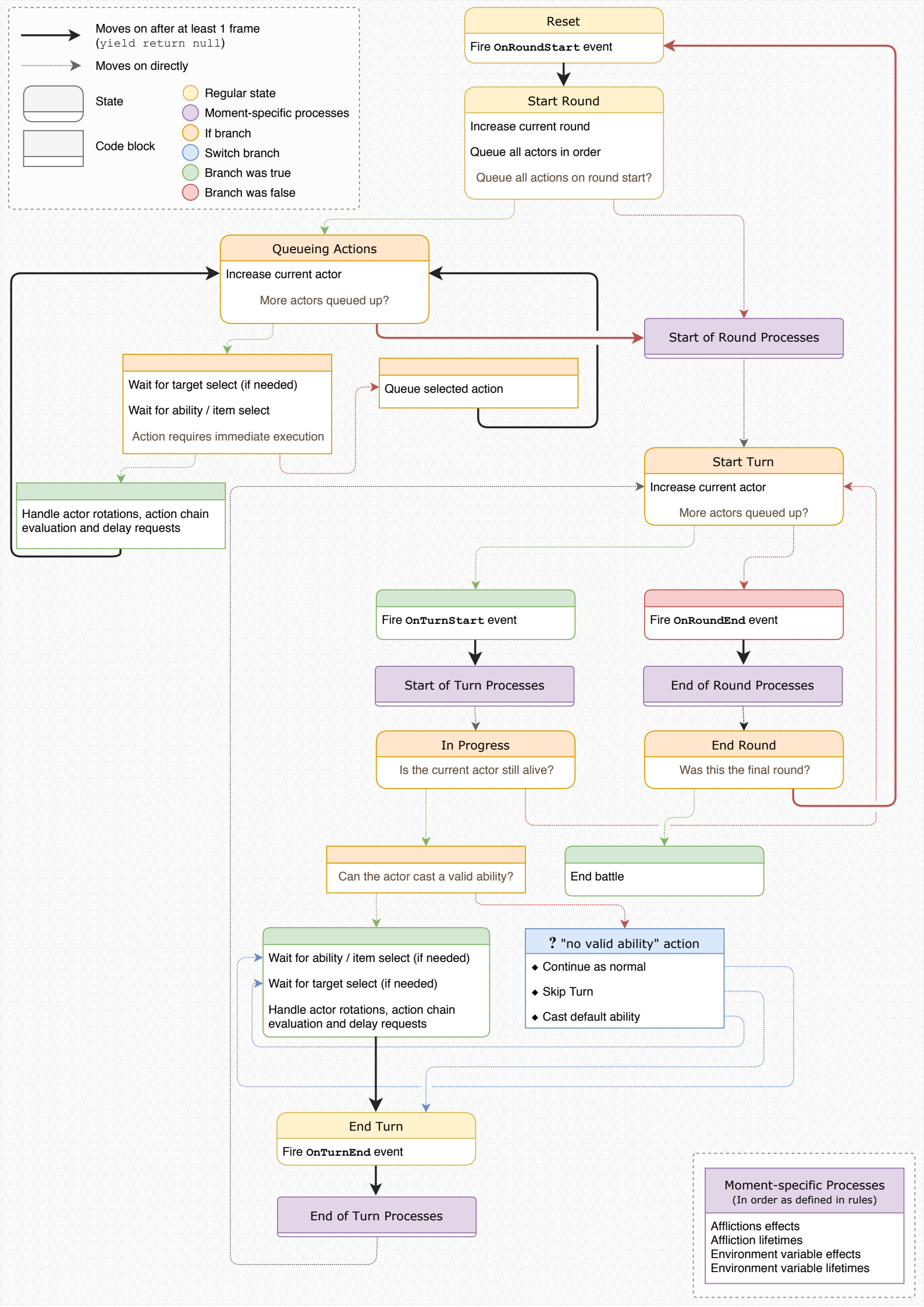
Doing so will fire the battle's `OnBattleEnd` event. This same event is fired when the battle's taken its maximum number of turns allowed, and will pass along the reason for ending the match. Your battle's manager object is also recommended to listen to this event. If the reason is internal (i.e.: `Battle.EndReason.OutOfTurns`) you can then handle the custom logic in determining a "fallback" winner.

Note: Do not call the `EndBattle()` method from inside this last callback, as this will create an infinite loop. Nobody likes infinite loops.

Main Battle Logic Loop

On the next page is a diagram showing (somewhat simplified) the main lifecycle of the **Battle** object's different states and stages.

The biggest difference in progression lies at the Action Select Moment rule. By setting this to `OnRoundStart` the battle has an extra queuing phase in which all of the actor's actions are decided before being executed. Setting it to `OnTurnStart` instead will let each actor decide as their turn comes up, executing the chosen action immediately after.


Moves on after at least 1 frame
(yield return null)
Moves on directly
State
Code block
Regular state
Moment-specific processes
If branch
Switch branch
Branch was true
Branch was false
Reset
Fire OnRoundStart event
Start Round
Increase current round
Queue all actors in order
Queue all actions on round start?
Queueing Actions
Increase current actor
More actors queued up?
Start of Round Processes
Wait for target select (if needed)
Wait for ability / item select
Action requires immediate execution
Queue selected action
Handle actor rotations, action chain evaluation and delay requests
Start Turn
Increase current actor
More actors queued up?
Fire OnTurnStart event
Fire OnRoundEnd event
Start of Turn Processes
End of Round Processes
In Progress
Is the current actor still alive?
End Round
Was this the final round?
Can the actor cast a valid ability?
End battle
Wait for ability / item select (if needed)
Wait for target select (if needed)
Handle actor rotations, action chain evaluation and delay requests
? "no valid ability" action
Continue as normal
Skip Turn
Cast default ability
End Turn
Fire OnTurnEnd event
End of Turn Processes
Moment-specific Processes
(In order as defined in rules)
Afflictions effects
Affliction lifetimes
Environment variable effects
Environment variable lifetimes

Events

As the main coordinator the `Battle` class holds a lot of different events, from dealing with battle flow and state to requesting user input.

Battle events	
OnBattleStart	Invoked when the battle is first started.
OnBattleEnd *Battle.EndReason reason*	Invoked when the battle ends.
OnRoundStart *int round*	Invoked when a new round is starting.
OnRoundEnd *int round*	Invoked when a round is ending.
OnTurnStart *Actor turnActor*	Invoked when a new turn is starting.
OnTurnEnd *Actor turnActor*	Invoked when a turn is ending.
OnTurnSkip *Actor turnActor,* *bool voluntary*	Invoked when a turn is skipped. If `voluntary` is `false` the actor was forced to skip somehow (i.e.: out of time, selected action no longer valid).
OnRoundStateChange *RoundState state*	Invoked whenever the internal round state changes.
OnAbilityResults *AbilityResults results*	Invoked when all effects of the cast ability have been processed and evaluated.
OnItemResults *ItemResults results*	Invoked when all effects of the used item have been processed and evaluated.
OnEnvironmentVariableSet *EnvironmentVariable envVar*	Invoked whenever an environment variable is set.
OnEnvironmentVariableUnset *EnvironmentVariable envVar*	Invoked whenever an environment variable is unset.
OnEnvironmentVariableProcess *EnvironmentVariable envVar*	Invoked whenever an environment variable is about to process its effects.
OnEnvironmentVariableDurationChange *int newDuration*	Invoked whenever an environment variable has it's active turns remaining changed.
OnEnvironmentVariableStageChange *int newStage*	Invoked whenever an environment variable has it's stage changed.
OnActorStartedParticipating *Actor actor*	Invoked whenever an actor starts participating and becomes an active member (i.e.: gets their own turn).
OnActorStoppedParticipating *Actor actor*	Invoked whenever an actor stops participating and becomes a passive member.
OnActorNeedsActionInput *Actor actor* *ActionInput input*	Invoked whenever a Player Actor requires input. `input` contains the callbacks needed to make a selection.
OnActorNeedsSingleTargetInput *Actor turnActor,* *TargetInputSingleActor input*	Invoked whenever a Player Actor requires a single target to be chosen. `input` contains the callback needed to make a selection, as well as a list of all valid targets.

Battle events	
OnActorNeedsActorsTargetInput *Actor turnActor,* *TargetInputNumActors input*	Invoked whenever a Player Actor requires (up to) a certain number of targets to be chosen. `input` contains the callback needed to make a selection, as well as a list of all valid targets.
OnActorNeedsGroupTargetInput *Actor turnActor,* *TargetInputGroup input*	Invoked whenever a Player Actor requires a Battle Group target to be chosen. `input` contains the callback needed to make a selection, as well as a list of all valid targets.
OnActorHasGivenAllNeededInput *Actor actor*	Invoked when a Player Actor has provided all required input.
OnActorDiedBeforeMakingMove *Actor actor*	Invoked when a Player Actor has had their HP reduced to 0 before they could execute their selected action.

Timing and Battle Delays

Overview

One of the greater pains of setting up a system like this is taking into account all the various amounts of animations, UI effects and other elements that might affect the timing of your battle.

ARES tries to take care of as much of these out of the box as possible, and offers helper methods for any unforeseen or external cases.

First, it is possible to request an additional timed delay before the battle progresses. This can be done by calling `Battle.RequestProgressDelay(delay, reason)`. The delay is measured in seconds, and the reason can be one of the following enum options:

Battle Delay Reasons	
Ability	Used for situation related directly to the execution of an ability. Use-cases might include variable-timed effects, or the result of a delayed battle effect. Ability delays are always granted.
Animation	Used by the **Actor Animation** component and for variable-timed animations or animated effects. For example, some effects might take longer if the enemy is further away physically. Animation delays can be disabled in the **Battle Rules**.
UI	Used for any UI effects. Typical use-cases might include HP or experience bars filling up, or a "Level Up!" message being displayed. UI delays can be disabled in the **Battle Rules**.

Timed requests start counting down as soon as they are made, and stack on top of the default wait times between progression (as defined in the **Battle Rules**).

A second way of delaying the battle from progressing is by requesting a delay lock. A delay lock prevents the battle from progressing until the lock is lifted. The method call for this is `Battle.RequestProgressDelayLock(requestor, reason)`. The reason is the same is before, and the requestor is the object (of type `BattleDelayElement`) making the request.

Each **Battle Delay Element** can only lock a battle for one reason at a time. You can derive any custom delay elements from this class, or you can choose to simply add it as a separate component. When choosing the latter, each component counts as it's own, unique lock.

When a **Battle Delay Element** is destroyed, any lock that may still have been in place will be released at the end of the frame.

Since multiple battles can take place within the same scene at the same time **Battle Delay Elements** need to be linked to the battle they intend to lock. The `Battle` class keeps a static list of all active battles (`Battle.ActiveBattles`), as well as a convenience getter for the active battle created most recently (`Battle.LastActiveBattle`). If your game only ever has one battle going on at a time, it's safe to just always link to the latter.

Enabling the component's `autoLinkToLastActiveBattle` property will cause it to try to link itself to the most recent active battle. This will happen in the component's `Start()` method, or the next frame if the `delayAutoLockByOneFrame` toggle is checked.

Actions and Action Chains

Overview

Abilities, **items** and **afflictions** are all **Chain Evaluators**. They each contain the ability to affect the battle and its actors in some way. These effects are referred to in ARES as **actions**.

Actions

Every **action** is one of the following types:

Action Type	*ChainableAction.Action*
Damage	Deals damage to the target.
Heal	Heals the target.
Buff / Debuff	Modifies a single stat on the target.
Afflict	Inflicts the target with a status affliction.
Cure	Cures one of the target's status afflictions to some degree.
Environment	Sets, unsets or modifies an environment variable.

Since it is common for a single in-game action to have multiple effects, these actions live within a sequential **Action Chain** list that can be used to build up your effects.

While each chain will receive one or more designated targets (either chosen by the player or determined in some other way), each individual action has the ability to override this target. This is especially useful when an effect has secondary actions that require a different target than the one that was chosen.

Actions can have one of three ways of determining power level:

Power Type	*ChainableAction.PowerMode*
Constant	A constant value.
Random	A random value between *power1* and *power2* (inclusive).
Formula	A value that is evaluated from a dedicated formula. Previously stored action results can be used from their identifier here.

Of these, the formula type requires some further explanation. Power formulas can make use of various macros and variables, and can be used to create complex interactions.

Power Formula	
Operators	`+` `-` `*` `/` `^` `()`
Functions	• `ABS()`: Absolute value • `MIN()`: Minimum value • `MAX()`: Maximum value
Macros	• `#D#`: A random value denoted in standard dice roll notation. The first number is the number of dice to cast, the second is how many sides each die has.
Built-in identifiers	• `[ACTOR]_HP`: The chosen actor's current HP • `[ACTOR]_MAX_HP`: The chosen actor's current max HP • `[ACTOR]_[STAT]`: The current value of the chosen actor's chosen stat Where: • `[ACTOR]` is either `CASTER` or `TARGET` • `[STAT]` is an actor's stat as named by it's Scriptable Object (see the Stats documentation below for further information on this)
Dynamic identifiers	• `[ACTIONTYPE][#]`: The amount of power processed by a previous action • `[ACTIONTYPE][#]_RAW`: The amount of power as initially calculated by a previous action Where: • `[ACTIONTYPE]` and `[#]` combine to form a previous action's identifier (see the Action Chain documentation below for further information on this)

Below are a few example formulas:

Formula	Behaviour
10 + 3D20	Has a base power of 10, plus the result of three 20-sided dice rolls (a combined value between 3 and 60).
50 * (CASTER_SPEED / TARGET_SPEED)	Has more power the faster you are compared to your opponent.
MIN(2 * CASTER_ATTACK, 120)	Has a power level of twice the caster's attack stat, capped at 120.

Action Chains

Action Chains are essentially just lists of **actions** to be executed in order, and be interrupted at specific points.

The results of each action is stored so that it can be used in subsequent steps. For example, an ability might first want to deal some variable amount of damage, then restore some percentage of that value back to the caster's health.

These identifiers are accessible through the format `[ACTION TYPE][#]` and `[ACTION TYPE][#]_RAW` where `[ACTION TYPE]` is the action type capitalized, and `[#]` is the number of times that type has occurred in the current list so far (starting at 1).

The *`_RAW` variant will store the result as initially calculated by the chain, whereas the plain version will return the actual effects (i.e.: amount of damage dealt or stages cured). The latter takes into account things like the target's defense stat, damage modifiers and the max amount of HP that would be taken or restored. Note that for actions that have multiple targets, each identifier will refer to the respective values for that particular target only.

Below are all the action type's representations:

Action Type	ACTION value	ACTION_RAW value
Damage	The final amount of damage dealt.	The base power to be sent to deal damage.
Heal	The final amount of HP restored.	The base power to be sent to restore HP.
Buff	The final number of stages the stat was buffed with.	The number of stages to try to buff the stat with.
Afflict	The stage at which the affliction was set if one did not exist yet, or the final amount it was modified by based on the affliction's "double set" behaviour otherwise.	The stage at which to try to set the affliction if one does not exist yet, or the final amount to try to modify it by based on the affliction's "double set stage" behaviour otherwise.
Cure	The final number of stages the affliction was cured of. A value of `-1` represents a full cure.	The number of stages to try to cure the affliction of. A value of `-1` represents a full cure.
Environment (set)	The stage at which the environment variable was set if one did not exist yet, or the final amount it was modified by based on the variable's "double set" behaviour otherwise.	The stage at which to try to set the environment variable if one does not exist yet, or the final amount to try to modify it by based on the variable's "double set stage" behaviour otherwise.
Environment (unset)	`1` if the environment variable was succesfully unset, `0` if not.	Undefined. Do not use this value. (It will evaluate as normal using the last values set from the Inspector before the set mode was changed to `unset`.)

In the case of the life-stealing attack example above, the first action in the chain would be a `Damage` type one, and thus gain the identifier `DAMAGE1`. The second action could then be of type `Heal` and use one of the following power formulas, depending on the type of behaviour you want. (Let's assume the attack action could deal up to 80 damage, but the target only had 50 HP left):

Formula	Behaviour	Result
DAMAGE1 * 0.5	Heal back 50% of the damage actually dealt to the target (50).	25
DAMAGE1_RAW * 0.5	Heal back 50% of DAMAGE1's calculated power level (80).	40

As each action has its own hit chance, this leaves one more problem to deal with. Namely, we only want the healing portion of our ability to trigger if the damaging action actually hit.

One option is to enable the `breaks chain on miss` option of the damage action. As the name suggests, this action missing its target will lead the action chain breaking and not being evaluated any further. This options then gives you full control over the healing action, being able to give it its own accuracy, timing duration and processing times.

The second option is make our healing action a child effect of our damaging one. When an action is made into a child effect it is treated as a direct consequence of the first non-child action before it. It will trigger as soon as the parent action does and will not add its own duration to the chain. A child effect will always trigger, can't miss its target and thus can't break a chain on its own. It will, however, obtain its own identifier and add its results to the list of evaluated values.

Action Tokens

In addition to the **Action Chains'** successive action identifiers, **Action Tokens** can be used as well. These are defined separately, and only get evaluated once – at the start of an item or ability's execution for `Constant` and `Random` values, and at the start of each individual target's execution for `Formula` ones.

Timing

Timing each action can get hairy quick, with things like animations, effects and UI updates to keep into account.

The 3D Battle example for instance has a life-stealing spear attack. It's completely reasonable to want to trigger the HP changes while the spear is still inside the enemy. This means that waiting for the animation to finish isn't an option. Ideally, our example ability would play out as follows:

> Entry animation ➟ Small delay ➟ Process both actions' effects at the same time (damage and life gain) ➟ Exit animation.

To accomplish this ARES has separated out the processing time from the total duration for each Ability Action. In addition, a base duration has been added to allow time for entry animations or other set-up work. The combined duration can therefore be found by adding together the ability's base duration and each individual action's duration.

All timing parameters are in seconds.

Timing paramaters	
Base Duration	A base delay that is applied before the first Ability Action is started. This can be used to play any foregoing animations e.g. This base duration can be overridden on a per-actor basis, from their Abilities list.
Action Duration	The total duration of each Ability Action. The Ability Chain will continue processing once this amount of time has passed. This value is not overridable, and thus the same for all actors.
Action Process Time	The normalized time at which the action's effect should begin processing.

However, this only covers the timing for the abilities themselves. The battle's progression time can be further influenced by that battle's rules' timings, as well as by **Battle Delay Elements** such as UI elements or custom objects spawned as the result of an ability. For more on this, see Timing and Battle Delays.

This is fairly complex example. A more accurate (depending on your needs) version of this can also be achieved using a more elaborate Animation Controller and some custom scripting.

You could for instance use two separate animations, with the first ending in a hold and needing a second animation parameter to transition. A script could then set this parameter either as or from a custom Battle Delay Element, by waiting for the battle's next continuation, or by hooking directly into the target(s)' HP bars or other elements and waiting for those to finish.

Battle Interactors

Overview

Items and **Abilities** are collectively referred to as **Battle Interactors**. They are functionally identical, though they are used in different contexts.

Targeting

Both items and abilities are used as actions by actors directly and require a use target. Depending on the type of interactor this target might be the same as the caster. First, the different target types:

Target Type	
Single Actor	A single actor that the interactor operates on. This is probably the most-used type of target.
Number of Actors	A set number of actors for the interactor to operate on. This amount can be set using the `Number of Targets` property that becomes available when this target type is selected. One typical use-case for this target type is abilities with AoE (Area of Effect), though the targets can be chosen directly and are not aware of in-scene positioning. If enough targets are not available (or passed through code) the interactor will operate on those that are instead.
All Actors in Group	This target type expects a Battle Group as input. It will then operate on all members of this group. Typical use-cases include mass healing of buff-type interactors.
All Actors	All actors in the battle will be affected by this interactor.

Where the Target Type determines the amount of actors and how they are selected, there are two properties that act as a filter on which actors are allowed as targets.

The first such filter is Valid Targets or Valid Target Groups, depending on the target type. The available options are `Self`, `Allies` and `Opponents` in their various combinations. A value of `None` can be used when the action chain only uses random targets. In this scenario, no hit test will be executed and the attack will always land. (Note that individual actions within the chain can still miss.)

The second filter is the Valid Target State, which can filter by alive or defeated actors only if desired.

Multi-turn actions

Some items or abilities might have powerful effects that require a build-up or cooldown period. A strong mana burst for instance might take a turn to charge before firing. For this purpose interactors have Preparation and Recovery properties. They have a set amount of turns, interrupt options and fields for custom messages.

If a preparation is interrupted before it is finished, the interactor will have its use cancelled. The messages can be used to display text to the user. Two tokens are allowed here:

Preparation and Recovery tokens	
ACTOR_NAME	The preparing or recovering actor's name
TURNS_REMAINING	The number of turns remaining until the preparation or recovery is complete.

Misc. Properties
There are two more properties that deserve a quick explanation.

First, the Base Duration is an additional timing that can be included before the action chain starts executing. This can be useful when you need to give your actors or environment some additional time to handle things like animations or in-game UI/ effects.

Lastly, Turn Towards Target signals to the battle system whether the actor should attempt to face the target. The final call for if and when this rotation happens is made based on the associated settings in the **Battle Rules**.

Actors also can not turn to face themselves. When multiple targets are chosen the actor will face the targets' averaged position (excluding itself).

Abilities

Overview
All actions that an actor can perform aside from items are classified as **Abilities**. Abilities can do anything from manipulating stats to dealing damage or inflicting status afflictions.
Because every game's different, special care has been taken to keep the out-of-the-box functionality as generic as possible, as well as making sure it can be easily expanded upon. For example, your game might call for a distinction between *spells* and *physical* attacks, or an *element* property. This should be added by modifying or extending the *Ability* class.

Events
Ability events all live inside the `Actor` class only. This has been done to consolidate this functionality and prevent having to setup unique handlers for each individual ability.

Ability events	
OnAbilityPreparationStart *Ability ability,* *string message*	Invoked when the actor has cast an ability that requires some turns of preparation. The message can be used to user display purposes.
OnAbilityPreparationUpdate *Ability ability,* *int turnsRemaining,* *string message*	Invoked when the actor's currently preparing ability gets closer to completion. The message can be used to user display purposes.
OnAbilityPreparationEnd *Ability ability,* *bool interrupted*	Invoked when the actor's currently preparing ability has stopped preparing. If `interrupted` is `true` the preparation failed and the ability will not be cast.
OnAbilityStart *Actor caster,* *Ability ability*	Invoked when an actor has begun casting an ability. Will be invoked even when the first action misses.
OnAbilityActionProcess *Actor caster,* *Ability ability,* *AbilityAction action*	Invoked when the ability action is ready to be processed (the step where the damage, buffs etc. are calculated and applied). The Battle system uses this event to trigger the actual processing.
OnAbilityActionEnd *Actor caster,* *Ability ability,* *AbilityAction action*	Invoked when the action has been processed and it's entire duration has passed. Will not be invoked if the action missed.

Ability events	
OnAbilityActionMiss *Actor caster,* *Ability ability,* *AbilityAction action*	Invoked when the action has been processed but did not land. Will be invoked on the ability's caster.
OnAbilityActionAvoid *Actor caster,* *Ability ability,* *AbilityAction action*	Invoked when the action has been processed but did not land. Will be invoked on the action's target.
OnAbilityEnd *Actor caster,* *Ability ability*	Invoked when the ability is done processing. This can either be because all actions in the ability's action chain have processed, or because the chain was broken.
OnAbilityRecoveryStart *Ability ability*	Invoked when the actor has finished casting an ability that requires some turns of recovery.
OnAbilityRecoveryUpdate *Ability ability,* *int turnsRemaining,* *string message*	Invoked when the actor gets closer to recovering from their previously cast ability. The message can be used to user display purposes.
OnAbilityRecoveryEnd *Ability ability,* *bool interrupted*	Invoked when the actor has fully recovered from their previously cast ability. If `interrupted` is `true` the recovery has been ended prematurely.

Items

Overview

While **Items** are extremely similar to **Abilities** they have been split up none-the-less.

Items can be consumed in order to get their action chain evaluated. They can be used on both the actor using it, as well as a separate target. Items have limited uses.

Because Ares items are extremely light-weight besides their battle-specific functionality it should be fairly straightforward to create Ares items from your own items if you are using them. Simply create a corresponding Ares item for each one and hook into the `Inventory.OnItemConsumed` callback to keep track of any changes that you may need to propagate back into your own system.

User mocking

It's common in games to use one actor's turn to instigate the use of an item on another. In some cases this is very straightforward, like actor A throwing a rock item at actor B. There are also cases however where this does not make sense. For instance, when the player user actor A's turn to select the use of a healing potion on actor B. In this case we would expect actor B to be seen as the item user, as well as the target. This becomes especially clear in interactor effects that want to spawn a prefab at the item's user for example, or status texts like "Actor A used a healing potion".

For this reason items have a `mockUserAsTarget` property. This forces the item's user to be set to the target actor.

Events

Item events all live inside the `Actor` class only. This has been done to consolidate this functionality and prevent having to setup unique handlers for each individual item.

Ability events	
OnItemPreparationStart *Item item,* *string message*	Invoked when the actor has used an item that requires some turns of preparation. The message can be used to user display purposes.
OnItemPreparationUpdate *Item item,* *int turnsRemaining,* *string message*	Invoked when the actor's currently preparing item gets closer to completion. The message can be used to user display purposes.
OnItemPreparationEnd *Item item,* *bool interrupted*	Invoked when the actor's currently preparing item has stopped preparing. If `interrupted` is `true` the preparation failed and the item will not be used.
OnItemStart *Actor caster,* *Item item*	Invoked when an actor has begun using an item. Will be invoked even when the first action misses.
OnItemActionProcess *Actor caster,* *Item item,* *ItemAction action*	Invoked when the item action is ready to be processed (the step where the damage, buffs etc. are calculated and applied). The Battle system uses this event to trigger the actual processing.
OnItemActionEnd *Actor caster,* *Item item,* *ItemAction action*	Invoked when the action has been processed and it's entire duration has passed. Will not be invoked if the action missed.
OnItemActionMiss *Actor caster,* *Item item,* *ItemAction action*	Invoked when the action has been processed but did not land. Will be invoked on the item's user.
OnItemActionAvoid *Actor caster,* *Item item,* *ItemAction action*	Invoked when the action has been processed but did not land. Will be invoked on the action's target.
OnItemEnd *Actor caster,* *Item item*	Invoked when the item is done processing. This can either be because all actions in the item's action chain have processed, or because the chain was broken.
OnItemRecoveryStart *Item item*	Invoked when the actor has finished using an item that requires some turns of recovery.
OnItemRecoveryUpdate *Item item,* *int turnsRemaining,* *string message*	Invoked when the actor gets closer to recovering from their previously used item. The message can be used to user display purposes.
OnItemRecoveryEnd *Item item,* *bool interrupted*	Invoked when the actor has fully recovered from their previously used item. If `interrupted` is `true` the recovery has been ended prematurely.

Afflictions

Overview

Afflictions are status conditions that live on actors. They contain action chains that can optionally be evaluated once per turn at one of the following moments:

Affliction events	
Start of turn	Evaluated when a new round starts. Afflictions are ordered by actor first (following the order in which they'll take their turn) and by time added second.
Start of afflicted actor's turn	Evaluated when the afflicted actor's turn starts, before they can select or make their move. Afflictions are ordered by the time they were added.
End of afflicted actor's turn	Evaluated when the afflicted actor's turn ends, after all of this move's item and ability effects have triggered. Afflictions are ordered by the time they were added.
End of turn	Evaluated when a round ends. Afflictions are ordered by actor first (following the order in which they'll take their turn) and by time added second.

Afflictions can also be used as status markers with no further functionality. In this case you can simply set the evaluation moment to `Never`.

Actors can only be afflicted by the same affliction once at a time. Trying to afflict an actor with the same affliction can either do nothing or modify its stage and/ or duration, depending on how you set up its "double set" behaviours. This can be configured per affliction.

Events

Affliction events all live inside the `Actor` class only. This has been done to consolidate this functionality and prevent having to setup unique handlers for each individual affliction.

Affliction events	
OnAfflictionObtain *ActorAffliction affliction*	Invoked when the actor obtains the affliction.
OnAfflictionStart *ActorAffliction affliction*	Invoked every time the affliction triggers, before any individual actions are processed. Will be invoked even when the first action misses.
OnAfflictionActionProcess *ActorAffliction affliction,* *AfflictionAction action*	Invoked when the affliction action is ready to be processed (the step where the damage, buffs etc. are calculated and applied). The Battle system uses this event to trigger the actual processing.
OnAfflictionActionEnd *ActorAffliction affliction,* *AfflictionAction action*	Invoked when the action has been processed and it's entire duration has passed. Will not be invoked if the action missed.
OnAfflictionActionMiss *ActorAffliction affliction,* *AfflictionAction action*	Invoked when the action has been processed but did not land.
OnAfflictionEnd *ActorAffliction affliction*	Invoked when the affliction is done processing. This can either be because all actions in the affliction's action chain have processed, or because the chain was broken.
OnAfflictionStageChange *ActorAffliction affliction,* *int newStage*	Invoked every time the affliction's stage changes. Will not be invoked if a stage change is attempted but the new stage is the same as the old one.

Affliction events	
OnAfflictionVariableDurationChange *Affliction affliction,* *int newDuration*	Invoked every time the affliction's remaining duration changes. Will not be invoked if a duration change is attempted but the new duration is the same as the old one.
OnAfflictionCure *ActorAffliction affliction*	Invoked when the actor is about to lose the affliction.

Environment variables

Overview

Environment Variables can be thought of like global variables, and are typically used for battle-wide effects like weather conditions or aura effects. They might be used to modify certain stats, change the power or duration of **items**, **afflictions** or **abilities**; or they could perform effects like damaging or healing every **actor** at a specific moment every turn.

Unlike all other types, environment variables use **filters** and explicit callbacks for its functionality, rather than an action chain. Since environment variables are so context-specific the implementation of their non-filter effects is left entirely up to the developer and will require a bit of coding.

Filters

Although most effects will require some coding, a very common use of environment effects is to modify power levels or to prevent the use of certain items or abilities. To accommodate this each environment variable has a list of **filters**.

Each filter operates on one power type (damage, heal, etc.) at a time, and has the ability to modify its power, and to make interactors of that type uncastable.

This power modification is made after all others, meaning actor effects like defense are taken into account first.

Callbacks

While you could check and process each variable manually from within the `Battle` class, the preferred method is to add your custom callbacks to the `EnvironmentVariableCallbacks` class.

You can choose to either add your callbacks at runtime by calling `EnvironmentVariableCallbacks.RegisterCallback()`, or edit the class file directly. While the former offers more flexibility and prevents you from having to deal with a modified source code, editing this file directly for effects that never change can help to consolidate this functionality and prevent forgetting to add certain callbacks at different times. If you are considering taking this route please make sure to read the section on Handling modified versions of ARES to gain an understanding of what this might mean for future updates and development.

If you decide to opt for the static approach, you can implement whatever processing methods you require inside the class. To have the method called, simply add your callback methods to the `callbacks` dictionary at the top of the file.

When adding any callbacks either way, it is important to note that the *key* in the `callbacks` dictionary is the `EnvironmentVariableData` object's name (this is *not* necessarily the same as the object's display name, and is case-insensitive). The *value* is an

`EnvironmentVariableCallback` object that houses `Set`, `Unset` and `Process` callbacks. All callback methods receive both the **Battle** is was called from and the **Environment Variable** itself.

Environment Variables can only be set on the same battle once at a time. Trying to set the same variable while it is still in effect can either do nothing or modify its stage and/ or duration, depending on how you set up its "double set" behaviours. This can be configured per environment variable.

If your effect takes some time to display and wants to halt the battle, you can call either `RequestNextRoundDelay()` or `RequestNextRoundDelayLock()` on the **Battle** object from your callback method. (For more information see Timing and Battle Delays.)

Events

Environment Variable events all live inside the `Battle` class only. This has been done to consolidate this functionality and prevent having to setup unique handlers for each individual variable.

Environment Variable events	
OnEnvironmentVariableSet *EnvironmentVariable envVar*	Invoked when an environment variable is first set.
OnEnvironmentVariableProcess *EnvironmentVariable envVar*	Invoked when the environment variable is about to be processed and have its `ProcessCallback` method called.
OnEnvironmentVariableDurationChange *EnvironmentVariable envVar,* *int newDuration*	Invoked every time the environment variable's remaining duration changes. Will not be invoked if a duration change is attempted but the new duration is the same as the old one.
OnEnvironmentVariableStageChange *EnvironmentVariable envVar,* *int newStage*	Invoked every time the environment variable's stage changes. Will not be invoked if a stage change is attempted but the new stage is the same as the old one.
OnEnvironmentVariableUnset *EnvironmentVariable envVar*	Invoked when an environment variable is first unset (removed).

Effects

Overview

All of the four objects above (**Abilities**, **Items**, **Afflictions** and **Environment Variables**, as well as the **Actor Components**, use **Effects** to set animation parameters, play sounds or instantiate objects at various times.

In the case of abilities, items and afflictions, their effects are only executed if they were triggered by an actor with the relevant `Actor*` component attached.

Animation Effects

Animation effects are used to set Mecanim animation parameters. These can be used however you see fit within your Animation Controllers.

When used on **Actor Animation** components you can select the parameter directly. If it's attached to an interactor it takes the parameter name as a string instead.

Note that this component is requires for any animation delays to work. If an interactor's animation effect requests a delay and is used by an actor without the component attached, the request will be ignored.

Audio Effects
Audio effects are very straight-forward. The play a chosen AudioClip at some volume after a set delay. Where relevant you can also set the sound to play at the position of the effect caster, target or the main camera.

Instantiation Effects
Instantiation effects are a little more involved, but still straight-forward enough. There are various options for indicating how and where an object should be spawned, and includes an option to parent the new object to a Transform in the caster or target's hierarchy.

Only objects that have an `InstantiationEffectObject` component attached can be spawned. You can choose to inherit from this class for your own effects, or just attach it besides it.

Inventory

Overview
To facilitate the use of items, Ares comes with two default types of **inventories**. These inventories can be created inside your custom battle manager and linked to either an entire **Battle Group**, an individual **Actor**, or both.

How you choose to present the various options to your players is entirely up to you and the needs of your game, but generally speaking group inventories would make sense for shared inventories like a player or enemy inventory, whereas actor inventories can be used to store things like enchantments or individually held items.

The two default inventories that come with Ares are a **Linear** and a **Stackable Inventory**. The **linear inventory** functions just like a flat list, with each item appearing in order whereas the stackable one allows for any item sharing the same Item Data *and* remaining number of uses to stack into a single Stackable Item. A common use case for this would be an inventory of consumables like potions, where the player might have a large number of them at any time.

One thing that's important to note is that the same `Item` object can not be added to the same inventory more than once. This does not mean that an inventory can't hold multiple items of the same type, just that they will be unique objects in your code. It is recommended to add items by using the `AddItem(ItemData)` overload to avoid any confusion. The `AddItem(Item)` overload does not add the object by reference, but rather treats it like a template and creates a cloned item instead.

Events

Inventory events	
OnItemAdd *Item item,* *int amount*	Invoked when an item is added to the inventory.
OnItemRemove *Item item,* *int amount*	Invoked when an item is removed from the inventory.

Item Groups
Item Groups are assets used to bundle various items together. They are meant for use with **Inventory Generators.**

Inventory Generator
The **Inventory Generator** is a convenience asset which can be used to quickly create inventories. It supports guaranteed items as well as picking from random ones and **Item Groups**. They can be set up from the Inspector, and generate inventories by calling `GenerateInventory()`.

Inventory Builder
The **Inventory Builder** is a convenience component that can generate inventories from **Inventory Generators**. When attached to an Actor it can then automatically assign it the generated inventory.

Stats

Overview
Nearly all turn-based games use various stats on their characters to create variation. Ares fully supports custom stats, but comes with and depends on three out of the box: **Attack**, **Defense** and **Speed**.

By default the **Attack** and **Defense** stats are used during an item or ability's damage calculation by multiplying the action's evaluated value by `attacker's Attack / target's Defense`. This default behaviour can be modified in `BattleInteractorData`'s `EvaluatePower()` method.

Speed is used by the **Battle Rules** as one of the options to determine the order at which actors get to make their moves.

To make updating and maintaining stats as painless as possible they have been implemented as Scriptable Objects in the `Resources/Stats/` folder. You can freely add and remove stats here, and they will automatically appear on each actor's inspector and in Action Chains' stat selection enums.

Additionally, each stat is available through code by calling `Actor.Stats[statName]` where `statName` is the name of the Stat object (case-insensitive). Note that this is *not* necessarily the same as the stat's display name. This was a deliberate choice, allowing you to change the way you present the stat to your users on the fly, without having to change all references in your code.

Unlike other Scriptable Objects used throughout Ares, like Items and Abilities, all Stats need to be in the Resources/Stats/ folder. This is because actors need to be able to load all stats and verify they have all of the required ones (and no old, since-deleted ones).

Events
Stat events all live inside the `Actor` class only. This has been done to consolidate this functionality and prevent having to setup unique handlers for each individual stat.

Affliction events	
OnStatBuff *Stat stat,* *int newStage*	Invoked when the stat's stage is increased. Will not be invoked if a stage change is attempted but the new stage is the same as the old one.

Affliction events	
OnStatDebuff *Stat stat,* *int newStage*	Invoked when the stat's stage is decreased. Will not be invoked if a stage change is attempted but the new stage is the same as the old one.

Battle Rules

Overview

The **Battle Rules** define various aspects of how the battle plays out and cleans up after itself. Every battle requires a single set of rules.

Progression and Timing Settings

First up are the **progression settings**.

Max Rounds allows you to set a maximum amount of rounds before the battle ends. If all the rounds have been played out the `Battle.OnBattleEnd` event will be invoked with the `EndReason.OutOfTurns` argument and the battle will end. Your custom battle manager can then catch this event and determine who wins (or call a tie) and handle any custom logic there.

Progress Automatically allows you to toggle between using automatic or manual progression from one state to the next. Note that if you use manual progression, timing settings and battle delays are no longer taken into account for you.

If automatic progression is enabled the **timing settings** are made available as well.

The Initial Battle Progress Delay is used to add a delay between the battle being started and the first round starting. (Note that even with this delay set to `0` there will be a single-frame delay.)

Next up are the Time Between * options. These are delays in progression applied after each respective action (moves, rounds, affliction effects, affliction duration adjustments, environment variable callbacks and environment variable duration adjustments.).

The Turn Timeout dictates the maximum amount of time a player has to select their actions and targets. If this is left at 0, unlimited time is granted.

Last are the Wait For * Events toggles. These correspond to the various Battle Delay Reasons (UI, animations and abilities) and allow to easily allow or disallow certain delays.

Action Settings

The action settings are where most of the settings relating to the flow of the battle live.

The Can Select * Without Valid Target settings do exactly what their name implies. With these settings enabled all abilities and items are considered valid. When disabled, any `validAbilities` or `validItems` lists passed by the battle will be pre-filtered to exclude any items or abilities that can't be used at the time of choosing. If a 0-target item or ability is sent back anyway an assert exception will be thrown.

The No Valid Actions Action is used to select what happens when no valid item or ability can be selected. This is the case when both:

1. Can Select Ability Without Valid Target is set to false and no ability has any valid targets, or the actor has no abilities; and

2. Can Select Item Without Valid Target is set to false and no item has any valid targets, or both the actor and its associated battle group have no items.

The various options are as follows:

No Valid Actions Action	
Continue As Normal	Progress the battle as always, waiting for input if needed. This setting is useful if you still want to show the default UI so the player can switch out characters or perform some other type of action the ARES is unaware of.
Cast Default Ability	Bypass the normal action selection flow and attempt to cast a default ability instead. This ability can be set in the Default Ability field that is otherwise hidden, and can be overridden on a per-actor basis in the `Actor` component. If the default ability can't be cast either the turn will be skipped and ended.
Skip Turn	Don't do anything and end the current turn.

Next, the Action Select Moment determines at what point the action to perform is chosen.

Action Select Moment	
On Turn Start	Each actor will choose their action at the start of their turn, then immediately execute it.
On Round Start	Add an extra stage at the start of each round where each actor needs to pre-plan their next move (in order). These actions are then queued and executed during their respective turns.

If the Action Select Moment is set to `On Round Start` an extra setting for the Item Consumption Moment will be shown.

Item Consumption Moment	
On Round Start	Use the item as soon as all actions are chosen, any before any abilities, regardless of actor sort or priority. If multiple items are used this round they are resolved in the same order as they were selected.
On Turn	Act just like abilities and queue use of the item for the actor's turn.
On Turn But Mark As Pending	Act the same way as the last but add the actor to the item's list of pending users. This allows you to filter out queued up items so that they can't be chosen twice for instance.

The next two settings here, the Invalid * Target Actions will only show when their respective Can Select * Without Valid Target is enabled, or when the Action Select Moment is set to `On Round Start`. Both of these scenarios can lead to no valid targets for the selected action being available when the action executes.

Invalid [Ability I Item] Target Action	
Select Random Target	Attempt to find a random valid target for the chosen action. This can succeed when a queued up action's target has since been defeated, but there are other valid targets e.g. If a new target cannot be found, the turn will be skipped and ended.
Skip Turn	Don't do anything and end the current turn.

Lastly, if a Turn Timeout has been set you can select the action to take when time does run out with the Turn Timeout Action setting.

Turn Timeout Action	
Cast Random Ability	Attempt to find a random valid ability to cast with random valid targets. If one cannot be found the turn is skipped instead.
Cast Default Ability	Attempt to cast the fallback ability (either from the actor or, if not specified there, the battle rules) with random valid targets. If this cannot be done the turn is skipped instead.
Skip Turn	Don't do anything and end the current turn.

Actor Settings

The Actor Sort is used to determine the turn order within rounds.

Actor Sort	
By Speed	Sorts the actors by their `Speed` stat, in descending order. (This uses .NET's default `Sort()` method. If two actors have the same `Speed` value, their order may differ between sorts.)
Random	Do a completely random shuffle.
None	Leave the actors in the order they were added to the battle.

Actor Can Face Target comes into play when an actor uses an item or ability that is marked to want to face the target.

Actor Can Face Target	
Always	Always turn the actor to face the target.
On Attack Hit	Only turn the actor to face the target if the item or ability's first action lands.
Never	Never turn the actor to face the target.

If Actor Can Face Target is not set to `Never` the Actor Face Speed setting is shown to set how fast an actor turns (in degrees per second).

Post-Battle Cleanup Settings

These settings deal with destroying any objects that may be created during the battle. Destroy selects which objects to dispose of, and the Destruction Delay adds a little time (in seconds) between the end of battle and the actual destruction.

Effects Processing Order

In some cases timing conflicts might occur. For instance, an affliction and environment variable might both want to apply their effects at the end of any given turn. This list determines in which order these effects should be resolved. Entries within each category are executed in the order they were added.

Creating actors from code

Overview

While it's nice being able to set up actors from the Inspector, this isn't always desirable; sometimes it makes more sense to create and set up your characters purely from code.

The example scene's `BattleManager` script shows a concrete example of how to go about this. Below is an abridged outline of which steps to take, and in which order.

Example

```
-
```

```
void SpawnActor{
      GameObject actorGO = Instantiate(characterPrefab);
      Actor actor = actorGO.AddComponent<PlayerActor>();

      /* First, create your actor's abilities and store them in an
       * Ability[]. Repeat for afflictions and an inventory if needed.
       * Next, create a Dictionary<string, int> to hold the actor's stats.
       * The key is the name of the stat as you would otherwise access it.
       * After that we can finally initialize our Actor by calling its
       * Init() method.
       */

      actor.Init("Actor name", currentHp, maxHP, stats, abilities,
                  fallbackAbility, afflictions, inventory);

      //  Once we've done that we can add any other components we may need.

      ActorAnimation animation = actorGO.AddComponent<ActorAnimation>();

      /* Before we can edit any of the components' callbacks we need to
       * initialize them.
       */

      animation.Init(allowDefaultAbilityAnimation, allowDefaultItemAnimation,
                     allowDefaultAfflictionAnimation, resetParamsAfterSet);

      /* We can then grab hold of any existing callbacks using the
       * component.GetXXXCallback(...) methods, or add new callbacks using
       * the component.AddXXXCallback(...) ones. Event and Ability
       * callbacks are added automatically and cannot be added manually.
       */

      ActorAnimationEventElement takeDamageCallback =
            animation.GetEventCallback(EventCallbackType.TakeDamage);

      // Lastly we can setup and enable the effects as desired.

      takeDamageCallback.Effect.SetAsTrigger("TriggerParamName");
      takeDamageCallback.Effect.SetAsAnimationDelay(animationLayer);
      takeDamageCallback.ignoreEffects = AnimationEventIgnoreFlags.Abilities;

      takeDamageCallback.Enabled = true;

      // Repeat this for all callbacks and components as needed.
      }
}
```

Extending the engine

When and Why

In general it would be recommended to build your own functionality on top of ARES. There are a lot of events to hook into, and many ways of modifying the battle state (where it makes sense). Following this path will generally be a lot easier as it requires very little knowledge of how the engine works under the hood and greatly minimizes the chances of breaking something inadvertently.

That said however, don't feel held back when extending the engine seems like the easier, cleaner or more maintainable way forward. ARES does not and can not account for everything, and some functionality might just make more sense internally. For example, you might want to prevent certain **Battle Groups** from using items. While this can be handled externally, it makes way more sense to just add a `canUseInventoryItems` bool to the `BattleGroup` class.

There is no hard line between right or wrong here, so use your best judgment and do what works for you and your project!

Handling modified versions of ARES

One thing to note here is that modifying the engine will mean you can no longer simply update to the latest version of ARES, as doing so will overwrite your local changes. It may also not be immediately obvious that your version of ARES is not the standard one.

As always, make a project backup before upgrading to a new version of Ares. Going into source control is way outside the scope of this document, but if you aren't already (and you want to modify the engine), looking into this is very highly recommended.

Modifying existing code

Ok, so you have decided to modify ARES a little. Below are some examples that are very likely to come up during development.

Hit testing

First, we'll take a look at the **Actor**'s `PerformHitTest` method. This method determines whether an action will land or not. There are three overrides (one each for abilities, items and afflictions). We will be focusing on the ability's for now. Let's look at the default implementation first:

Actor.cs

```
public BattleInteractorData.HitStatus PerformHitTest(Actor caster, AbilityData
ability, AbilityAction action){
      if(Random.Range(0f, 1f) <= action.HitChance){
            return BattleInteractorData.HitStatus.Hit;
      }

      return BattleInteractorData.HitStatus.Evade;
}
```

As it stands, the method simply generates a random number between 0 and 1. If it's less than or equal to the action's hit chance, the attack lands.

You can make this logic as complex or as simple as required. Below is a modified example. In it, we assume that:

- Actors derive some kind of evasiveness factor from their and their opponent's speed stats.
- Actors have a new `isGhost` variable.

- Actors who are ghosts always evade their next incoming attack.
- Actors have a new `equipment` variable.
- Equipment objects have some kind of `ModifyEvasion()` method that can affect the evasion result (and then handle any side effects like equipment wear-out e.g.).

Here is what this might look like in practice:

Actor.cs

```
public BattleInteractorData.HitStatus PerformHitTest(Actor caster, AbilityData
ability, AbilityAction action){
      // If we're a ghost, evade no matter what and unghost.
      if(isGhost){
            isGhost = false;
            return BattleInteractorData.HitStatus.Evade;
      }

      // A modifier based on the actors' speeds.
      float evasivenessModifier = Mathf.Min(1f, caster.Stats["speed"] /
Stats["speed"]);

      // Our equipment's effects
      equipment.ModifyEvasion(ref evasivenessModifier);

      // Apply our final evasion value to obtain our result
      if(Random.Value <= action.HitChance * evasivenessModifier){
            return BattleInteractorData.HitStatus.Hit;
      }

      return BattleInteractorData.HitStatus.Evade;
}
```

Mana or limited-use abilities

Next, let's look at adding mana or limited uses to abilities.

First, it is important to note that all **Abilities**, **Items**, **Afflictions** and **Environment Variables** consist of two parts.

The first part is the ***Data** component. This is the **Scriptable Object** that you see and edit in the Inspector when you create a new interactor from the `Add > Ares` menu. This contains all the static information, like min and max stages, power levels, action chains, etc.

The second part is a wrapper class of the same name, minus the `Data` part. So in the case of abilities, there will be an `Ability` class that holds a reference to the appropriate `AbilityData` object. This wrapper class is where all the per-instance information lives.

So armed with this knowledge, we can now get an idea of how to implement our new ability's requirements!

Mana cost would likely be a single, static property, so it can live inside the `AbilityData` class. Next, we need someplace to check against our actor's mana. Like all Battle Interactors, `AbilityData` derives from the `BattleInteractorData` class, which conveniently houses a `CanUse()` method. However, we likely want this mana requirement to be present only for abilities, and none of the other interactors. As such, we'll choose to override the `CanUse()` method inside the `AbilityData` class instead.

First, let's add our new mana property:

AbilityData.cs

```
public class AbilityData : BattleInteractorData<AbilityAction> {
	public int ManaCost {get{return manaCost;}}

	[SerializeField] int manaCost;
}
```

We create a serialized field so that we can edit it inside the Inspector, and then a public getter to make this value accessible to the outside.

Next, I'm going to assume you've already added a `Mana` field to the `Actor` class, and instead focus on the `Ability` class' logic.

AbilityData.cs

```
public class Ability : BattleInteractor<AbilityData, AbilityAction> {
	[...]
	public override bool CanUse(ActorInfo user, ActorInfo target){
		if(user.Actor.Mana < Data.ManaCost){
			return false;
		}

		return base.CanUse(user, target);
	}

	public override void OnUse(Actor caster, Actor[] targets){
		base.OnUse(caster, targets);

		caster.Mana -= Data.ManaCost;
	}
}
```

Here we simply perform our mana check in our `CanUse()` method as discussed above. If we fail, we return `false`. If we do have enough mana, we'll call the base implementation for any further checks and return that result instead.

Next we override the `BattleInteractor.OnUse()` method to actually deplete the casting actor's mana. If you would prefer to keep this functionality completely within the `Actor` class however, you could have that subscribe to its own `OnAbilityStart` event and handle the logic there instead.

Like this mana cost, a maximum amount of uses per ability could be implemented very similarly. One would simply add some kind of `maximumUses` property to the `AbilityData` class, a `usesRemaining` property to the `Ability` one, and perform a similar check as above (`usesRemaining < Data.maximumUses`).

Power evaluation and immunities

Lastly, we will take a look at how to change the way items' and abilities' power is calculated. Again, we make a few assumptions. Namely, that:

- Actors and ability actions both have a new `element` variable.
- Actors are immune to elemental attacks of a certain type, stored in `Actor.immunityElement`.
- Actors whose element is `Light` will double their healing and curing abilities.
- Damage-type actions have a 5% chance of being a critical hit, dealing 50% more damage.

The damage calculation logic lives inside the `BattleInteractorData` class, in a method called `ApplyPowerModifiers()`. This method is marked virtual and can be overridden to handle logic exclusively to either items or abilities, but for our purposes we will be changing the base implementation. Note that `interactorTarget` and `actionTarget` can refer to two different actors. This happens when an action from the chain is set to use a non-specified actor. Also be aware that the `result` value is passed by reference, so there's no return value. Any changes to this value are directly reflected in the result; do not use it as an intermediary variable.

BattleInteractorData.cs

```
public class BattleInteractor<T1, T2> : ChainEvaluater where [...]{
	[...]
	protected virtual void ApplyPowerModifiers(Actor caster, Actor
interactorTarget, Actor actionTarget, AbilityAction action, ref float result){
		// First we test for immunity. If the action's target
		// is immune we set `result` to 0 and stop.
		if(action.element == actionTarget.immunityElement){
			result = 0f;
			return;
		}

		switch(action.Action){
			case ActionType.Damage:
				// Default stat implementation
				result *= caster.Stats["attack"].Value /
actionTarget.Stats["defense"].Value;

				// Test for critical hit
				if(Random.value <= .05f){
					result *= 1.5f;
				}

				break;
			case ActionType.Heal:
				// Check to see if healing should be doubled.
				if(caster.element == Element.Light){
					result *= 2f;
				}
				break;
			case ActionType.Afflict:
				break;
			case ActionType.Cure:
				// Check to see if curing should be doubled.
				if(caster.element == Element.Light){
					result *= 2f;
				}
				break;
			case ActionType.Buff:
				break;
			case ActionType.Environment:
				break;
		}

	}
}
```

When removing existing fields or properties, note that this will likely break the corresponding custom editor or property drawer. You can follow the errors given in the inspector to remove the offending field drawer.

Aditionally, when adding new properties they will by default show up at the bottom of the inspector. You can generally add these to the custom drawers without much trouble if you wish to do so.

Locating the code to edit

While the examples above may seem fairly straight-forward, they do require some prerequisite knowledge of where to look. But what if you have no idea where to even start?

The most effective way will likely be to "follow the code". If you want to include some kind of affliction immunity, your best bet might be to locate the `Affliction` class and use your IDE to find all references to it. Alternatively, you could start by looking at the `Battle` class, performing a search for the word `affliction` and just following the logic (jump to the source of possibly relevant methods) until you reach your destination.

In general, try to go as "deep" as possible. For example, don't implement the same change to chain evaluation for each interactor type inside the `Battle` class, but rather try and find a common point inside the `ChainEvaluator` class to edit instead.

Gotchas

While Ares tries to be as intuitive as possible there may be a few behaviours or ways of doing things that might not be immediately obvious. Below is a list of things to watch out for while developing your game:

- **Events are fired right before their effects are evaluated:** This may seem a bit odd at first – especially in some situations – but it has been a deliberate choice to keep this ordering consistent. Firing events this way allows custom scripts to hook in as early as possible, and allow for their own look-backs. For example, the `Actor.OnHPChange` event sends the actor's new HP as its parameter. Your event listener can then read out the actor's current HP to determine by how much it has changed and in which direction.

- **There are some dead logic frames:** Some effects may be delayed by a single frame where one might not expect it. This is most often done to provide an opportunity for custom scripts to request progress delays. For example, when starting a new round the `Battle.OnRoundStart` event is fired, followed by a single frame delay. That way, if your script requires some time to show UI elements or special events for instance, it can delay further continuation of the battle during that frame from within its own event handler.

- **Importing ARES will throw a bunch of warnings:** If you are using the .NET 4.x Scripting Runtime Version in your project you might notice a bunch of *CS0649* warnings that look like this:

  ```
  [script][line]: warning CS0649: Field '[name]' is never assigned to, and will always have its default value [value].
  ```

  This is because the 4.x runtime uses the new Roslyn compiler, which doesn't recognize fields marked with `[SerializeField]` as being accessible to the outside through the editor. Unfortunately however, this is still the preferred and recommended workflow within Unity.

  If these warnings get in the way a lot you can disable them by creating a plain-text file called `csc.rsp` in your root `Assets` folder, and have it say `-nowarn:0649` inside it.

Note however that this is a global solution only, and thus silences all uninitialized field warnings throughout your project, including legitimate ones.

Acknowledgements

Special thanks to the following people for providing assets used in the example scene.

- **Sound effects:**
  - ***RPG Sound Pack (CC0) by Ben ("artisticdude"):***
    https://opengameart.org/content/rpg-sound-pack
  - ***3 Hoar's thunders (CC0) by mangiabambini:***
    https://freesound.org/people/mangiabambini/sounds/169083/
  - ***Misc. sound effects (CC0) and editing by Kasper den Ouden***
    https://www.linkedin.com/in/kasper-den-ouden-5b8b1384/
- **Models:**
  - **Athena and Ares created from Sintel Lite (CC BY 3.0) by Ben Dansie:**
    https://www.blendswap.com/blends/view/7093
- **Miscellaneous:**
  - **Promo art chain icon (Flaticon Basic License) by Vectors Market from www.flaticon.com:**
    https://www.flaticon.com/free-icon/link_321834