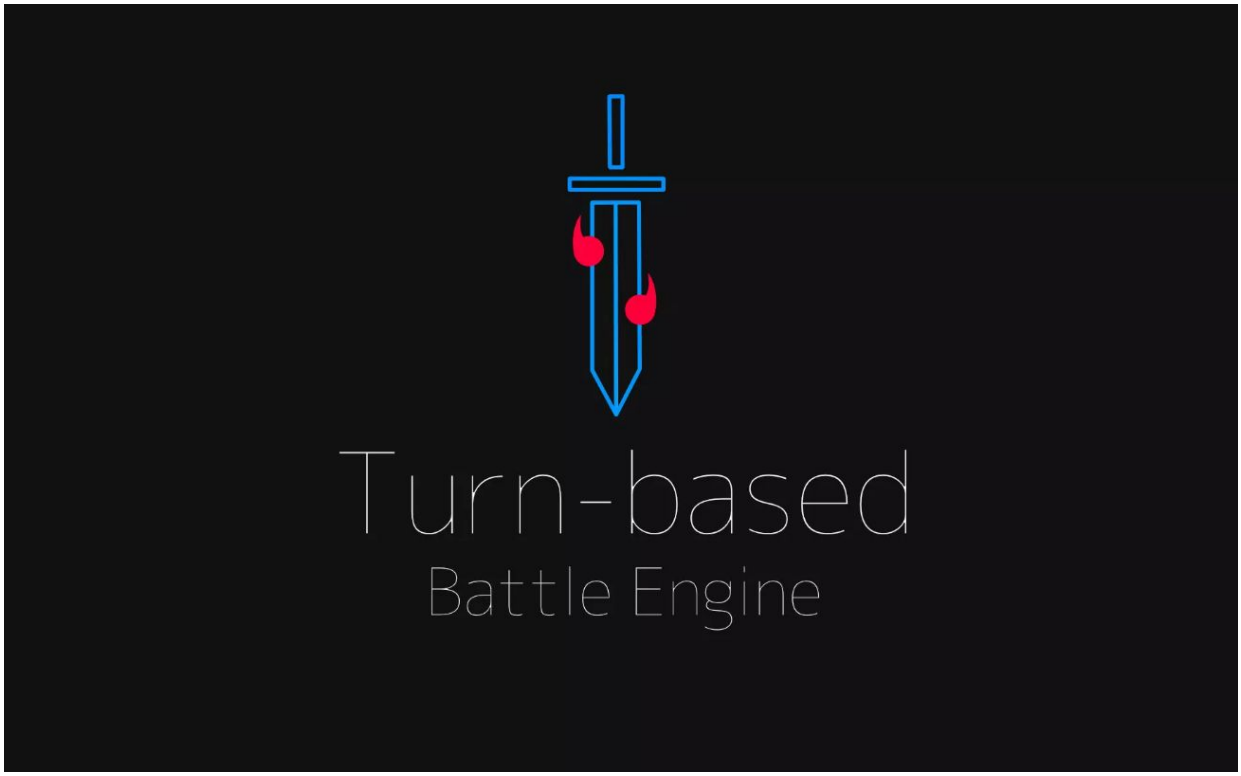
Turn-Based RPG Battle Engine 2D

Documentation



Thank you for purchasing Turn-based RPG Battle Engine!
If you feel like it, consider leaving a review. It only takes a minute and helps me out a lot!

Feel free to contact me at any time with any questions / suggestions.

Email: cinationproject@gmail.com
Unity connect: Link

Introduction

Being designed to assist in the development of custom turn-based battle systems, this asset facilitates the creation and modification of skills, items, battlers etc., by incorporating systems that allow the developer to focus on their project-specific functionality. The engine is incredibly flexible, making modifying any components of the default systems easy, as existing functionality is implemented as separate, interconnected components that can be effortlessly plugged in or out. All in all, the asset is designed to be as developer friendly as possible, while synchronously

giving the developer the freedom to implement new systems and modify any aspects of the existing ones.

Note: Throughout the documentation, I will be referring to various scripts. Each script is well commented, and I would highly recommend reading through the comments to accommodate yourself with the technical side of the asset.

Running the Demo

To run the demo, open the “scenes” folder, load the “SampleScene” and hit play. You can navigate the menus by using WASD, or the arrow keys to move, and the “space” and “enter” keys to select various options. Alternatively, the mouse can be used instead. Once one of the sides has zero remaining team members, the battle will end.

The project hierarchy looks as follows:

1. **Camera**: By default, only one, unmodified, camera (the main camera) is present on the scene.

2. **Game Manager**: The game manager object is essentially an empty game object which hosts all the asset’s scripts (we will be taking a look at all the scripts later on), and other components, such as two audio sources (one for the background music and the other one for additional sounds).

3. **Canvas**: The canvas hosts all the player UI. It is divided into the following sections:

4. **Footer**: The lower part of the UI. Character lists, actions, action description and the rest of the core UI is generated under dedicated sections of the footer.

5. **Body**: The body is the upper part of the UI. Player health labels, attribute changes (such as damage) and other fluid UI elements are generated under the body section.

6. **Outcome**: Used to display the outcome of the battle when either of the teams wins. Feel free to remove this section entirely if it does not suit your project.

7. **Combat area**: The combat area hosts the background art, the FX area, battler spawn points and sprites. The engine allows custom creating and assignment of spawn points. Each team can have any number of spawn points. The system will automatically restrict the maximum number of spawned characters based on the number of available spawn points for that specific team (we will look at battle initiation later). In other words, if you try to spawn 5 battlers but only have 3 battler spawns for the enemy team, only 3 of the characters will be spawned. As previously mentioned, feel free to remove and add any number of spawn points for each team, if each team has at least one spawn point. The

FX area is used to spawn various visual FX on top of characters (acts similarly to the "body" area of the canvas).

8. **Event system**: Just the default event system that comes with the canvas. Ignore it.

Structure

Before getting into project setup or any technicalities, it is very important to understand the general structure of the asset, code-wise. To do so, we shall look at the "GameManager" object and the collection of scripts that it accommodates.

Database: The database contains information about items, character and skills. You can view a simplified version of the database by clicking on the "Simple view" button at the very top of the component. By default, however, it is divided into separate sections for each one of the entities.

a) **Characters**: A list of all characters. Each character can be configured as follows:

   i. Name: The name of the character.
   ii. Id: A unique character id.
   iii. Description: Character description.
   iv. Icon: Character's icon.
   v. AI functions: A list of functions to be executed (from BattleMethods.cs) if auto battle is enabled or the character belongs to the enemy team. We will take a closer look at function configuration and invocation later.
   vi. Animation Controller: Each character should have a dedicated animation controller with animations specific to that character. We shall take a closer look at animations later.
   vii. Character attributes: Each character can have any number of attributes. Attributes can include things such as health, mana etc., and each attribute should have a unique id. By default, the attribute with id 0 should always be the character's heath.
   viii. Items: A list of all items available to the character. This list contains attribute ids which link to the Items list (at which we will look at next), and the quantity of the item currently in possession. Please note that if a character has x0 of the item, the item will not appear when the player accesses the items section during battle.
   ix. Skills: A list of all skills available to the character. Each entry of the list should be a skill id linking to the Skills list.
   x. Is active: A boolean which indicates whether the character is active (can battle) or not. When a character's HP reaches 0, by default they become inactive until they are healed.

b) **Items**: A list of all in-game items. Each entry is configured as follows:

i. Name, id, description: Name, unique id and description of the item.
ii. Turn point cost: The engine implements a turn point system. Each "action" has a turn point cost associated with it. We will take a closer look at the system and how to configure it later, but essentially this value represents the amount of turn points that is going to be subtracted from the currently available turn points. Once the turn points reach 0, the next character's turn will start.
iii. Functions to call: A list of functions to call from BattleMethods.cs when the item is used. Like mentioned before, we will take a closer look at function invocation later.

c) **Skills**: Although most parameters are identical to those of items, what distinguishes skills is the quantity variable being replaced with an "unlocked" boolean. Unless a skill is "unlocked" it will not appear in the corresponding section during battle.

Battle Manager: The battle manager is responsible for most of the engine's functionality. A series of coroutines are run within the script in order to monitor and apply changes to various aspects of the system. In other words, this script hosts all the engine's core functionality. The battle manager should be configured as followed (inspector-wise):

1. Active player team: A list of character IDs (Database -> Characters) to be spawned on the player team. As mentioned before, if the number of characters in this list exceeds the number of available spawn points, a portion of the characters, equivalent in number to the one of spawn points, will be spawned.

2. Active enemy team: A list of enemy team character ids.

3. Starting character id: The id of the starting character. You can set this id to -1 and the system will automatically assign the first player team character as the string character. However, if you choose a specific character (from either team) as the starting character, the team to which the character belongs to will initiate the battle.

4. Starting Music Id: The id of the music clip (from ObjectDB.cs which we will look at later), which will be played at the start of the battle.

5. Music Source Index: The index of the audioSource (attached to GameManager) that will play the music. The engine allows for multiple audio sources. By default, two audio sources are attached to the Game Manager object, one for the main soundtrack and one for secondary audio. The first audio source has index 0, the second one - index 1. If you add more audio sources, each consequent audio source will have the index of the previous audio source plus 1 as index.

6. Background Sprite Id: The id of the background sprite (in ObjectDB.cs - Sprites).

7. Auto Battle: If turned on, the AI Functions of every function on the scene will be triggered upon start. The player can use the “Auto” button located on scene to toggle this variable.

8. Max Turn Points: The maximum amount of turn points available per turn. We will take a closer look at the turn system later.

Object DB: A collection of object references. In order to maintain a tidier overall structure, all object references are and should be located under ObjectDB. Any object under ObjectDB can be accessed by any other scripts via class instance reference (ObjectDB.core.ObjectName). Beside on-scene objects, prefabs and other elements should also be stored and accessed through ObjectDB. For instance, “audioClips” a list of all audio clips and their respective ids, can be accessed from any other script in order to retrieve the desired audio by its id.

Function DB: In order to eliminate redundancies, all the functions used by more than one script should be hosted in FunctionDB. Any of the latter can be easily accessed via instance reference (FunctionDB.core.functionName), just like with ObjectDB.

Class DB: ClassDB is essentially a database of all classes used across the engine. This script is used like a library and should be included in any script to use any of its classes. This script can be included by adding the “using ClassDB;” at the upper part of the script.

Battle Methods: This script hosts all the methods to be invoked during action execution. When creating new skills or items, any custom functions to be executed by the entity in question should be added to BattleMethods.cs. As we’ve previously seen, each item and skill is accommodated by a “FunctionsToCall” list. The latter is essentially a list of all functions that will be called once the entity in question is interacted with. These functions can only be in BattleMethods.cs (as mentioned before we will look at how function invocation works, later).

Battle Gen: A collection of methods used to generate UI, battlers etc. The “Main Actions” List is essentially the list of all options the player sees when a character’s turn begins. Each action has a name, description and a list of functions (From BattleMethods.cs) to be triggered once it is selected.

Function Invocation

As we have seen from the previous section, custom function invocation is a big part of the engine. One of the things that make this asset so flexible, is the ability to invoke any custom functions when interaction with items or skills or even running character A.I.

Let us start by looking at how a “FunctionsToCall” list element is configured:

1. **Function name**: The exact name of the function to be called from BattleMethods.cs. For instance, if we were to create a skill from which a character takes damage to their health, at some point we would need to call the “changeTargetAttribute()” function to change the health attribute of the selected target. We will take a closer look at how functions can be chained to create skills and items, but for now let us focus on this example function solely. To call the latter function, first we would need to specify its name, thus writing “changeTargetAttribute” in the “FunctionName” field.

2. **Parameters array**: A list of parameters passed to the action. The engine allows you to have any number of booleans, string, int and float parameters for any function. In the case of “changeTargetAttribute”, the function itself has only two parameters: attrId and value. The first parameter is an integer, while the second one is a string. To pass values to these parameters via our inspector, we need to expand the parameters list to two items and write “int:0” in the first field and “string:+10” in the second one (these are simply examples of course). The reason why we need to specify the parameter type is because of how the system works. Essentially, all parameters passed through our inspector, are consequently passed to a function that decodes string into objects of certain types. The latter technique allows us to easily pass these custom parameters to our function without the need to have separate arrays for each type of parameters. In other words, the “parameters array” simply takes strings of the form “type:value” as parameters, and it is then used to decode and pass these parameters forth.

3. **Wait for previous function**: If this boolean is true, the engine will wait for the completion of the previous function in queue before executing the one in question.

4. **Is coroutine**: If the function in question is an IEnumerator instead of void etc., it is vital to set this variable to true, as coroutines are invoked differently from other functions.

Now that we have looked at the basic configuration of a “FunctionsToCall” element, let us take a more in-depth look at how this works.

First, let us assume that we have a skill with a list of functions (FunctionsToCall) configured. In other words, we have a list of functions and their parameters ready. So, what happens next?

Initially, the “FunctionsToCall” list is copied to the battle manager’s “functionQueue”. The function queue is essentially the list of all functions to be executed. Afterwards, a “methodCaller”(under BattleManager.cs) coroutine is started for each function in the queue. The method caller, as the name implies, is responsible for invoking the functions in the functions queue one by one. If the “waitForPrevious” variable is marked as true, the method called will wait for the previous function to be completed before jumping to the next one.

But how exactly does the method caller know when a function is completed?

Essentially, each method in the function queue has an “is running” variable. The state of the latter is what dictates whether the method caller will stay idle (I.e. the previous function is still running) or continue to the next function.

Important: The way the engine works, to achieve maximum flexibility when developing your assets, you, yourself must indicate when a function has finished execution. At the very end of each function under “BattleMethods.cs” you must include the following line:

BattleManager.core.setQueueStatus("NAME OF THE FUNCTION", false);

What this line does, is that it calls a function that sets the running status to “false” when called. As mentioned before, such an approx allows you to fully control and set the “break point” for each function.

If you are interested in the inner workings of this entire system, I highly recommend reading through the source code, as it is thoroughly commented.

Character A.I. Functions vs FunctionsToCall

The elements (I.e. functions) of both lists pass through the method caller when called upon. The difference is that the A.I. functions list is executed when the character in question belongs to the enemy team or “Auto battle” is enabled. On the other hand, “FunctionsToCall” is only executed when the player interacts with the item or skill which hosts that list.

Note: Although the engine comes with some very basic sample A.I., it is up to you as the developer to create the A.I. for your characters.

Note2: By default, the “SampleAI” function is the only function called for each character.

Creating function chains

Now that we have an idea of how functions are invoked, it’s time to take a look at how to create a proper function chain. For this example, let’s look at how the “Attack” skill is structured.

It is important to understand that in order to achieve maximum flexibility when creating skills and items, it is highly recommended to avoid highly interconnected functions. In other words, it is way more efficient to have a separate action for each single character, rather than one action that incorporates everything.

The “Attack” skill’s function chain looks as follows:

1. Select Target: A function that initiates target selection.

2. Wait for Selection: A function which only marks itself as not running once the player has finished selecting the attack targets (although may seem redundant, one look at the code will clarify how important this function is).
3. Toggle Actions: Disables actions (skills, items etc.) while the skill is executing.
4. Play Animation (first occurrence): Play walking animation.
5. Move To Target: Move the character towards the selected target.
6. Play Animation (2nd): Play attack animation.
7. Play Audio once: Play attack sound.
8. Change Target Attribute: In this case, we change the targets' health.
9. Wait: Wait for a set amount of time before continuing (essentially giving enough time for the sound and attack animation to play).
10. Rotate Character: Rotate character to face backwards (in this case).
11. Play Animation (3rd): Play running animation again.
12. Move Back: Move back to spawn point.
13. Rotate Character: Rotate character again.
14. Play Animation (4th): Play idle animation.
15. Toggle Actions: Re-enable actions.
16. Subtract points: Subtraction turn points.

Although this may seem overly complicated at first glance, such an approach allows to eliminate a huge amount of redundancy. When you have a certain amount of tested a working component, designing a new skill may only require the development of new components. This way, most skills will consist of mainly tested, already existing components which makes the development way easier since the margin for error during development is much smaller. Of course, the engine does not limit you to this approach. If you wish, you may have a single function for each skill, although not recommended.

Note: You can take a closer look at each function by inspecting the code and reading the relative comments.

Note 2: Although obvious, each of the previously mentioned functions is configured specifically for the skill in question.

All in all, designing function chains is simple and how your skills will work is completely up to you. The engine, by default, provides all the functions necessary to create any basic skill, however I highly recommend you develop your own functions in accordance to how you envision your skills.

Turn Point System

Throughout the documentation, so far, there have been multiple mentions of the turn point system. However, how exactly does it work?

This system aims at restricting the number of actions that a player can take per turn. The use of this system is completely optional; however, it is highly recommended for turn based games.

On each turn start the amount of turn points is set to the maximum amount of turn points available (set via the Battle Manager). Whenever we want certain actions (such as skills or using item) to subtract a certain amount of turn points, all we need to do is to include the "subtractTurnPoints" function into the function chain of the action in question, and specify the amount of points we would like to subtract.

By default, whenever the amount of turn points reaches 0, the turn on the character in question ends. It is important to note that the player can end their turn without the need to use up all their turn points.

Attributes

As mentioned before, each character can have a list of attributes, each with its unique id and values. However, it is highly recommended that the attribute ids for each character are consistent. In other words, if one character has a "Mana" attribute with id "2", all other characters with a mana attribute should also have it associated with id "2". If a character does not have a mana attribute, the id "2" should not be used by any other of their attributes.

This system will be reworked in future builds.

Animation

Since animation is a key part of any battle system, understanding how it is incorporated within the engine is very important.

First of all, if you are not familiar with 2D animation, I highly recommend visiting this tutorial on the official unity website.

It is important to understand that animator parameters can be used as conditions to trigger animation transitions. For example, by creating a parameter "idle" and assigning it to an "A->B" transition, we will be able to make the transition occur whenever idle is true.

Generally, instead of using the animations themselves, the engine uses transition booleans in order to trigger any animations. This implies that all animations that should be able to transition between each other, should be connected within the animator with the appropriate parameters set to the respective transitions. Generally, I highly recommend familiarizing yourself with the unity's animation system before continuing.

In any function chain you can play a given animation by calling the “playAnimation” function and passing the name of the animator parameter, that is supposed to trigger that animation, as a parameter to the function. The “displayFX” works in a similar way, as parameter names are used to play animations on the spawned “FXPrefab”s.

Creating a new character

To add a new character, simply create the character in the **Database**, and add the ID of the created character to either the **Active Player** list or the **Active Enemy** list.

As mentioned before, characters are based on animation controllers. You can add an animation controller to the character’s record in the database. If you are unsure about creating this component, please visit the **animation** section.

Battle Initiation

By default, the asset is designed to provide all the functionality to carry out a proper encounter, after battle has been already initiated. However, let us assume you are developing an RPG, in which battle should be initiated upon enemy encounter. How do we achieve such an effect?

By navigation to Scripts -> Optional, you will find a script called “VariableInitiator”. This script is designed to persist even when a new scene is loaded. If this script is on the scene attached to any game object, and our battle scene is loaded, the script will automatically transfer all its variables to the ones located under BattleManager and start the battle sequence afterwards.

In other words, in your main project, you can assign all the necessary variables to the Variable initiator upon enemy encounter, then load the battle scene, and the script will take care of transferring all the variables to the battle scene’s BattleManager, and starting the combat.

However, as mentioned before, you are free to take any approach you like to the combat initiation in your game.

Note: Combat is by default initiated by calling the “initiateBattle” method.

Outcome

Upon victory the “outcome” object, located under canvas, is enabled, and the id of the victor team is displayed. Since each turn-based game may treat defeat and victory differently, it is up to you as a developer to decide what will be the “consequences” of any of the latter occurrences. To do so, please navigate to BattleManager.cs, locate the “turnManager” method and add your code at the end of the segment labeled “Outcome”. The integer “victor” can be used to determine which side has won. If the integer has value 0, the player team won, otherwise the enemy team was victorious.

Setting up your project

Assuming that you are familiar with previous sections of the documentation, and have already added all your custom methods to the “BattleMethods.cs” script, navigate to the game manager object and set-up your skills, characters and items in the “Database” script. Once done, head to the “BattleManager” script and customize the values (player character list, enemy list, starting character id etc.) to your liking. It is worth mentioning that the latter step can be skipped if you are planning to use “VaribleInitiator.cs” prior to initiating combat (as described in the “Battle Initiation” section).

Conclusion

All the above systems will be improved and reworked in future builds. The latter alterations will most likely stem from user feedback. Thus, please do not hesitate to message me personally for any additions / modification you would like made to the asset.

End of Documentation.