# Turn Based Battle system

Thank you for purchasing the TBB kit. This kit was designed to be extendable into any kind of turn based system you choose to make. Since the main goal was to be as customizable as possible this system was designed to let as many components as possible operate as autonomous as possible. If not for touch support requirements, for instance, absolutely any part of the GUI could safely be removed and the entire game would still function 100% the same. In fact, if you design an alternative touch input system for your game, you still can…

This tutorial documentation is going to start you off with a very high level overview of how this kit works and will then gradually drill down into more specifics so you can understand what each component in this kit does.

## 1. High Level Overview

### The fundamental design

The goal was to create an event driven Data-Control-View model.
What this means is that the entire system can essentially be broken down into 3 parts:

1. DATA - All the game's data (with very few exceptions) all reside in one single location, the TBBData component
2. VIEWS - All GUI elements simply fetch whatever info they need from TBBData to be functional and displays it on screen. Interaction with the player is kept to the bare minimum
3. CONTROL - Each game type has a class that defines how that game type works. Any data it needs it fetches from TBBData and any data it changes it saves back to TBBData. It then calls whatever functions it needs to call and does whatever it is it wants to do using the data and when it is done it just tells the game that it is done with whatever it was doing. At this point the VIEWS pick up on this change and update themselves accordingly.

This is the key thing to understand in this system. All game functions should reside inside their appropriate classes and not interact with the GUI directly. Instead, it should interact with the data only and just indicate when it is done. All GUI elements will themselves interact with the data directly and update themselves accordingly.

Functions get their values from TBBData, do what they need to do and save data back to TBBData. That's where it ends
VIEWS will get their values from TBBData and display themselves as required but should not update TBBData since that is the job of the game mode's functions, exclusively.

### Event driven

Every action you take triggers an event to fire or will result in an event being fired at the end or both. The system was designed to work 100% without a GUI at all so as long as you can keep track of what state you are in and what buttons are active at the current point in time then you can run everything from opening credits to victory/game over without a GUI.

Events tell the system when to move from one state to the next and VIEWS simply listen for these events to be triggered and update, show or hide themselves based on which event got triggered.

### A simple example:

> When selecting which character you wish to configure you navigate through your options using the arrow keys. When you press the left or right arrow a function is triggered which fetches the value of 'selected_character' from TBBData and then either subtracts or adds 1 from or to the value, respectively, constantly checking if the character in the new location is available to configure and continuing to modify 'selected_value' until it either finds a suitable

character or determines there is no valid character to select while going in that direction.

If it determines a new character could not be found it simply returns and nothing happens. If it found a new character then it simply writes the new value of 'selected_character' back to TBBData and then triggers the OnCharacterSelectionChanged event.

At this point the VIEWS that care about that event being called now update themselves. In this case each view will check to see which character it represents and will check to see which character is selected by checking which character in TBBData.ActiveFaction.Participants is located at array entry 'selected_turn'. If it determines that it is not the selected character then it disables it's bouncing hand pointer. If it determines it does represent the selected character it enables it's bouncing hand pointer.

## Logic flow and managers

By far the most important part of this system is the game object you create onto which you place the components for the desired actions. Based on the type of game you are making you might want some features but not others. Each game type is defined by the combination of components you place on this core game object. To start you off the kit includes prefabs for 2 type of games.

The first is based on the classic Final Fantasy series where two teams in stationary positions fight each other in a your-turn-now-my-turn setup. The second version sees you able to traverse the board. This time your location on the board and the attack you choose determines your targets for you. More game modes will follow or you can create your own.

Secondly is the GUI. This is separate from the actual gameplay as each element is an object unto itself, as described in the previous section. The GUI is how you tell your players what they are doing and here you can add or remove components to suit your taste. For mobiles, though, these will respond to touches and will thus most likely have meaning. Select what components suit your game's needs.

The flow of the game is determined by the combination of components you drop onto the first game object but some features are simply mandatory and to make sure you don't skip any (or force you to drag an endless amount of components onto the game object) these derive from one another and means you only need to drag one component onto game object. The full relationship between these components will be described a little later on in this document.

In order to create completely different game modes you need only find the component where your code differs from the existing code and then create replacements for the script from that point all the way up the food chain to the final derived class. Unless your game is vastly different this code will remain mostly unchanged and require mostly copy-pasting to create your custom game type. A few functions will be changed a little and then, of course, you can add any additional code you feel you might need.

For instance, for the Final Fantasy mode I removed the ability to reposition your characters on the battlefield and made the states simply skip the state where you can change your position on the grid. In the EA version I left that code in place and while the majority of the code stayed the same, one function now goes to a different state when it is done and the new state goes to the state that the FF code would have gone to. In addition, when going backwards through your configuration the one state skips the repositioning state and goes one state further back. With that minor tweak I either enable full grid traversal or no repositioning at all.

## Artificial Intelligence

Fighting the computer wouldn't be much fun if the computer can't fight back. To that end each game mode simply has to create it's own class to define how to determine an attack and how to determine a target. This class with these two functions must derive from TBBEnemyAI and this class will automatically register itself with TBBData so that the different game modes can still access the relevant AI code. Let me just summarize that again :

1. Create a new enemy AI script deriving from TBBEnemyAI
2. Create the two functions the class must contain
3. Drop that AI script onto your game object

Done. Custom AI implemented for a custom game type. For the included game types the files are grouped together under the relevant subfolder under the GameModes folder.

## 2. Controls

Configuring your character's actions is done via a state machine. The different states include: CharacterSelect, NewLocationSelect, AttackSelect, AttackRefinement, TargetSelect

The kit runs on a virtual gamepad design where we assume you are using a joypad with 4 buttons, 4 bumpers, 2 thumbstick buttons (axis use not currently supported), start, select and the D-pad. For each state that the character configuration might be at you define a function to be called whenever any or all these buttons are pressed. If you do not define a function for a specific button then it is assumed that you do not intend for that button to do anything during that state. As such, you only define functions for the buttons that you need on a state by state basis rather than having to worry about what every single button does at every single second during the game.

The kit supports preset button configurations for the keyboard and 8 game pads but doesn't include a visual editor for you to configure them with. How you define your button config screens is up to you. See MBSInputManager.cs in the RPG folder for details on how this system works.

In order to use it, though, you need only set mbsInputManager.current_input to a value between and including 0 to 8 using the SelectInput() function and it will use the button presets you set for that particular controller.

**The default buttons are:**

| KEYBOARD VALUE | | ENUM VALUE |
|---|---|---|
| KeyCode.Return | | A |
| KeyCode.Backspace | | B |
| KeyCode.C | | C |
| KeyCode.Space | | D |
| KeyCode.Alpha1 | | L1 |
| KeyCode.Alpha2 | | L2 |
| KeyCode.Alpha3 | | L3 |
| KeyCode.Alpha4 | | R1 |
| KeyCode.Alpha5 | | R2 |
| KeyCode.Alpha6 | | R3 |
| KeyCode.UpArrow | | DUp |
| KeyCode.RightArrow | | DRight |
| KeyCode.DownArrow | | DDown |
| KeyCode.LeftArrow | | DLeft |
| KeyCode.LeftControl | | Select |
| KeyCode.Escape | | Start |

## 3. Derived classes

Certain classes are derived from one another to ensure they are not excluded when adding components to the game prefab while also removing the need to import them one at a time. The dependencies will be listed first, in order of final to base, followed by a short description of what each one does.

I will be using the Final Fantasy game mode scripts for this example:

**TBBInputsFF : TBBGameModeFF : TBBAGameMode : TBBInputManager : MBSInputManager**

MBSInputManager is a Singleton class that derives from MonoBehaviour.

### 1. TBBInputsFF

Here you need only override the Start() function and declare what button calls what function on a state by state basis.

Example:

```
SetStateAction( ECharSelModes.SelectCharacter, mbseButtons.A, ChooseCharacter );
```

Example explained:

1. Which of the states are we defining a button for?
2. Which button are we interested in?
3. What function should be called when this button is pressed?

### 2. TBBGameModeFF

This is where you define custom actions for your specific game mode. Here you will either write new functions, override base class functions or copy paste code from other game modes and just tweak it to suit your game mode.

The following functions are required to be in this class:

```
void GetSelectableTargets();
void ChooseCharacter();
void SelectAttackType();
void ConfirmRefinement();
```

### 3. TBBAGameMode

This contains the bulk of the game mode's logic and can be seen as the actual base class of every game mode. You will most likely not change anything in this and rather override some of these functions in your own class.

### 4. TBBInputManager

For the most part this is merely a convenience function. It contains the SetStateAction function you used earlier and simply shortens the syntax from "TBBData.SetStateAction" to just "SetStateAction". Since you might have quite a few entries in that script this just saves you a bit of typing.

Most importantly, though, while the PollInputs() function in MBSInputManager checks to see if a button was pressed, this function checks to see if any action was set to trigger for the current state if that button was pressed, and of course to then call that function if set.

Simply put, this is a super small class that is as important as it is small and yet there is nothing for you to ever need to change here so in almost all cases you are free to forget this class even exists.

### 5. MBSInputManager

This class loads and saves your button configurations and scans for the relevant buttons being pressed during the relevant state then triggers the event that causes your defined actions for that button to be run. This class is explained in great detail in the text file placed alongside it.

# 4. GUI Components

The goal with the GUI components, as mentioned earlier, was to have each one do nothing more than show something on screen. For touch input games it is practical to make them functional also but doing so simply means that you should make clicking a GUI element simply do whatever the "enter" or "A button" would do and then letting the relevant event cause the relevant code to be run.

**The GUI elements are NOT to be used to run code itself!**

The relevant game mode scripts perform the game logic and indicates when the GUI can be updated so the point of the GUI elements is just to display whatever they are meant to display. For this reason you will find that there are many GUI scripts in this kit and when you go to any of the arena scenes you will see that the canvas is filled with GUI objects.

This is not something for you to be worried about but rather something to enjoy. Since each component is merely cosmetic (mostly) it means that you can add or remove components to the GUI without breaking anything. Simply decide which elements make sense for the battle system you are using and add them to the canvas. Everything else, delete them or just don't add them. That simple.

For example:
In the Final Fantasy style arena you will notice that you can select the targets you want to attack and that as you select one or the other there is a super imposed graphic of the selected target displaying in the corner of the screen. Also, there is a window that tells you how many targets the current attack can have and how many targets you currently have selected as well as what buttons to press to select/unselect/confirm your selections.

In the Enchanted Arm style arena the attacks themselves select the targets for you meaning you never need to select them manually. Since you don't select them manually there is no need to show the super imposed character of the target you are considering choosing nor is there a need to show you how many targets you can select and what buttons to use to do so. Thus, in this arena I simply deleted those two game objects and moved on with the rest of the project.

### Utility components

During development it was useful for me to see what was going on behind the scenes. Stuff like "What game state is currently active?" "What character selection phase are we at?" "Where was the character originally and where do I intend to place him?"

All these things helped me in determining when to take what action and to debug issues during designing the game modes so I created simple GUI components to show these values on screen. I am leaving them in the project for you to use if you decide to create your own custom game mode but apart from that they can be safely ignored. I have placed them in a folder called Utilities so you will know which prefabs are important and which you can ignore or delete entirely if you so choose.

## 5. Quick overview of how to use this system

1. Inside your main game scene, instantiate a BattleLauncher component and configure it with all the settings you want. This includes the party members on both factions sides.
2. Optionally (and demonstrated in the demo scene) store your character's position and rotation in the current scene
3. Load a scene that has the BattleSystem prefab of your choice already in place (Final Fantasy style, Enchanted Arm style or your custom battle system) along with a canvas with all the GUI elements you want to display for that mode
4. The BattleLauncher from the previous scene will now copy over your settings to the BattleSystem prefab, spawn the battlefield and kick start the battle.
5. When the game is done it will trigger a victory or a lost event that you can hook into. In the demo scenes:
    a. When you loose I take you back to the scene you came from (if that is the scene you chose)
    b. When you win I load a screen where you can add your own logic to do o loot drops and award XP or whatever else you want to do. It also includes a button to take you back to the scene of your choice
6. If you did step 2 using the class I included for this purpose then your character will be returned to the scene and placed exactly where he was before.