BATTLE AND MINIGAME GUIDE

This document details how battles and minigames work and interact with each other in code. This should help anyone who wants to create a new minigame or change the way the battle system works in any way.

OBJECTS IN BATTLE SCENE

These are the objects in the scene Battle.unity, and what they are responsible for. The inner workings of each script will be covered in the Game Loop section of this document.

Minigame Controller

The minigame controller is designed to control transitions into and out of the minigames. It houses a Minigame Control script, which does exactly this.

Minigame Window

The minigame window functions as a screen on which the minigames are projected. It contains a Mesh Renderer component with a render texture called Minigame View as a material. This render texture displays everything in the separate minigame scene, allowing players to play the minigames. It also contains a Minigame Window Controller component. This component causes the minigame window to expand into view from the attacking character, and then retract back into that character when the minigame has ended.

Battle Control

The battle control contains a Battle Controller component, which controls the core of the game loop, cycling between different characters in the correct order and allowing them all to attack correctly. It also removes characters that have run out of health and ends the battle when a side has won.

Belligerents

These are the characters and enemies that take part in the battle. Player controlled characters (formally referred to as protagonists) and enemies (formally referred to as antagonists) are all children of this empty game object.

Protagonists

Protagonists are a series of objects referring to player-controlled characters, which are all children of the Belligerents game object. A protagonist contains a Sprite Renderer, showing the character (pixel art) and a Battle Stats component. Battle Stats controls each belligerent's attack, defense, and health, and processes the way these values change and how those changes are displayed on screen. It is VITAL that the protagonists' game objects' names are the same as the characters' names.

Antagonists

Antagonists have the same components as protagonists, but additionally have an Enemy AI component, which controls the different ways that enemies choose who they attack, as well as controlling whether an enemy can currently be selected as a target or not. Antagonists also contain a Box Collider 2D, which is used as a trigger, so the game can know which enemy the player chooses to attack by clicking on them. They also have a target indicator as a child, so players know when they can choose an enemy to attack, but this is subject to change, as all UI is temporary right now.

GAME LOOP: EARLY STAGE

The majority of this document will detail what happens in the code every time one character makes an attack. When the player finishes attacking, the code will move on to the next and repeat the cycle. This section refers to the early part of the game loop before a distinction is made between whether a protagonist or antagonist is made.

Starting a Battle

There is currently no system in place to trigger a battle from outside the scene, but battles start automatically when Battle.cs is loaded as of now.

Battles start with the function BattleController.AdvanceTurn(). This function is mostly in charge of setting and abiding by turn order.

Choosing Whose Turn It Is

BattleController.AdvanceTurn() will call BattleController.SetCurrentAttacker(). This function looks at who just took a turn, and, based on that, sets the current attacker. It sets the game object of the next attacker as BattleController.currAttacker, and sets their battle stats as BattleController.currAttackerStats.

BattleController.SetCurrentAttacker() also relies on two public bools, BattleController.proFirst and BattleController.alternateTurns. BattleController.proFirst sets whether the first attacker of the battle is a protagonist (true) or an antagonist (false). BattleController.alternateTurns sets the turn order in terms of switching teams; if this value is true, then the turn order alternates between protagonists and antagonists. If it is false, then every protagonist takes a turn, then every antagonist takes a turn, and so on. The default values are BattleController.proFirst = true and BattleController.alternateTurns = false.

PROTAGONIST TURN

When a protagonist has been chosen to take their turn, BattleController.AdvanceTurn() will call AllowTargetSelection().

Player Chooses An Enemy To Attack

BattleController.AllowTargetSelection() runs through all the enemies who haven't been defeated and for each one, calls EnemyAI.SetSelectionPhase(true). This function changes whether or not a given enemy can be selected for attack, and, if so, activates relevant UI to signify as such. Here, every opposing enemy is made available for selection.

Players click on an enemy in order to attack them. This calls EnemyAI.OnMouseDown(), which stops making that enemy available for selection to attack and then calls BattleController.RecordTarget(the selected enemy's battle stats). This function formally records the enemy the player chose to attack under the variable BattleController.enemyTarget, which is of type BattleStats. It also runs through each enemy and calls EnemyAI.SetSelectionPhase(false), making them unavailable for selection for attack. It then starts the coroutine BattleController.StartMinigame().

Loading a Minigame

BattleController.StartMinigame() starts the process of loading a minigame by calling MinigameWindowController.Expand(currAttacker.transform.position). This causes the minigame window to appear by expanding outward from the position of the attacking character to the middle of the screen. It then starts the minigame itself by loading MinigameControl.StartMinigameVisual() immediately, allowing the minigame's start state to look correct as soon as the player sees it, and calls MinigameControl.StartMinigameGameplay() when the window is fully expanded so that the gameplay of the minigame will start when the player is ready and no earlier.

MinigameControl.StartMinigameVisual() has a few purposes. Firstly, it loads a new, additive scene, which contains the minigame using the function MinigameControl.LoadMinigameScene(scene), which also moves the minigame control object to the minigame scene. The StartMinigame() function also finds the enemy that's being attacked and the current attacker's current level of tiredness using an Information object, which contains all information relevant to story and stats that moves between scenes. Tiredness is an important value in battles. Each character starts with a tiredness level of zero, and every time they attack successfully, their tiredness increases by one. The tiredness is reflected in minigames in a way that makes sense (e.g. more tired characters may move slower, have a harder time lifting heavy objects, are less agile, etc.), and should generally make minigames more difficult with more tired characters. Tiredness is designed to encourage players to play as a wider variety of characters, since they'll be motivated to play as less tired characters, which they've played as less in the course of the game, thus creating a more varied and enjoyable experience.

MinigameControl.StartMinigameGameplay() allows the active portion of the minigame to start, triggering all events that are either caused by the player or require quick responses from the player. Once this is done, the minigame can start.

Running a Minigame

Each minigame should be run by one class that functions as a control, and this class should inherit from a Minigame class. The Minigame class comes with some useful public variables, and functions:

- Minigame.tiredness (float) keeps track of the attacker's tiredness, and should be used in every minigame in a meaningful way.
- Minigame.enemy (enum enemyType) keeps track of the type of enemy being attacked, and should also affect the minigame in a meaningful (more than just cosmetic) way, so as to add more variation to the same minigames as the players progress.
- Minigame.readyCosmetic (bool) signifies whether or not the minigame is visible and has all information required to be set up visually. Once it is true, Minigame.StartCosmetic() will be run automatically. You should not use this variable, as its only purpose is to trigger Minigame.StartCosmetic() at the correct time, which happens automatically.
- Minigame.readyGameplay (bool) signifies whether or not the minigame window has fully expanded, and thus is ready for gameplay to begin. Once it is true, Minigame.StartGameplay() will be run automatically. You should not use this

variable, as its only purpose is to trigger Minigame.StartGameplay() at the correct time, which happens automatically.

- Minigame.End(MinigameResult result) must be called to end a minigame. It takes in a MinigameResult, which is an enum with three types: MinigameResult.won, MinigameResult.Lost, and Minigame.criticalHit. Every minigame should have a win state (MinigameResult.won), which inflicts damage to the targeted enemy in the main battle scene, a lose state (MinigameResult.Lost), which inflicts no damage, and a super win state (MinigameResult.criticalHit), which inflicts double the normal damage. Every minigame should make sure all three of these results are possible.
- Minigame.StartCosmetic() is a function which should be treated as a Start() function, but only load visual elements of the minigame (e.g. make sure the correct enemy is present, start idle animations, make sure all game elements are in the correct locations), but nothing that the player would want to respond to quickly (e.g. evasive movements of enemies, environmental changes, physical changes to any characters) or any player-triggered events (e.g. moving, firing weapons). When you declare this script in any class inheriting from Minigame, be sure to specify that it is a protected override void!
- Minigame.StartGameplay() is a function which should be treated as a Start() function, but only starts all events in the minigame players would want to respond quickly to and allows for the start of player input. When you declare this script in any class inheriting from Minigame, be sure to specify that it is a protected override void!
- Minigame.OnTimeUp() is a function that is called any time a timer runs out in the minigame. Every minigame comes with a timer, because it's vital that every minigame will automatically end at some point no matter what the player does or doesn't do in order to keep the flow of the battle going.
- Minigame.Start() is a function that should NEVER be used, as it may require/process information before the minigame is ready. Anything that would ordinarily be in a Start() function should be put in the StartCosmetic() or StartGameplay() scripts, or in the case of Start() functions on other MonoBehaviours in other minigame objects, should be triggered by these functions.

Ending a Minigame

As previously stated, minigames are ended by calling Minigame.End(result). This function moves the Minigame Control back to the main battle scene, unloads the minigame scene, and calls MinigameControl.ProcessMinigameResults(result). All that script does is retract the screen by calling MinigameWindowController.Retract(result).

MinigameWindowController.Retract(Minigame.MinigameResult result) does the opposite of MinigameWindowController.Expand(); it shrinks the minigame window to a size of zero and moves it towards the character who just attacked. When that is finished, it starts the coroutine FireProjectile(currAttackerStats, enemyTarget, bool critical) if the player did not lose the minigame (see Game Loop: Late Stage). If they lost, the turn ends, and BattleController.AdvanceTurn() is called to move on to the next character's turn.

ANTAGONIST TURN

Choosing A Victim

The following takes place right after BattleController.SetCurrentAttacker() selects an antagonist as BattleController.currAttacker. The distinction in programming logic is officially made when BattleController.AdvanceTurn() starts the coroutine BattleController.EnemyAttack(), which then calls EnemyAI.Attack() for the enemy that is currently attacking.

EnemyAI.Attack() is mostly responsible for choosing which protagonist will be attacked. There are a few strategies in place, but more are likely to be added. These are the strategies that currently exist:

- TargetStrong targets the protagonist with the highest attack value
- TargetWeak targets the protagonist that would require the fewest attacks to be defeated
- TargetRandom randomly chooses a protagonist to attack

Every enemy has a public array of strategies called EnemyAI.strategies. The strategies are listed in order of priority, starting with EnemyAI.strategies[0] having the highest priority. So the first strategy in the list will be employed, and if the strategy results in a tie, then the next strategy in the array is used as a tiebreaker, and so on, until a tie no longer exists. If there is still a tie, TargetRandom is used, which cannot result in a tie.

Example #1: An enemy werewolf (Attack 1, Defense 0, Health 5) is about to attack. The three protagonists it is facing are Mio (Attack 10, Defense 0.5, Health 7), Nitesh (Attack 10, Defense 0, Health 14), and Matthew (Attack 2, Defense 0, Health 1). The werewolf's EnemyAI.strategies = [EnemyAI.EnemyStrategy.TargetStrong, EnemyAI.EnemyStrategy.TargetWeak]. Which protagonist will the werewolf attack?

First, we run through the first strategy, EnemyAI.EnemyStrategy.TargetStrong, which means the werewolf will target whoever has the highest attack value.

Mio's attack = 10
Nitesh's attack = 10
Matthew's attack = 2
Since Mio and Nitesh are tied for the highest attack, the werewolf will attack one of them. Which one will be determined by the next strategy in EnemyAI.strategies.

Next we run through EnemyAI.strategies[1] = EnemyAI.EnemyStrategy.TargetWeak, which means the werewolf will attack whoever can be eliminated in the fewest hits.

When the werewolf attacks Nitesh, it will attack with a power of 1. Since Nitesh has no defense, he would take all that damage. Therefore, taking out Nitesh would require

(Nitesh's Health) / (werewolf's attack)
= 14 / 1
= 14 hits

When the werewolf attacks Mio, it will still attack with a power of 1, but since Mio has a defense of 0.5, that means she will be protected against half of the attack's total power, meaning she'd only take 0.5 damage from each attack. Therefore, taking out Mio would require

(Mio's Health) / (werewolf's attack * (1 - Mio's defense))
= 7 / (1 * (1 - 0.5))
= 7 / (1 * 0.5)
= 7 / 0.5
= 14 hits

Since it would take the same number of hits to eliminate either of them, we need more strategy to figure out who the werewolf would attack.

We've reached the end of EnemyAI.strategies, so we run the strategy TargetRandom. This means either Mio or Nitesh would have an equal chance of being attacked. Therefore,

There is a 50% chance of the werewolf attacking Mio, and a 50% chance of the werewolf attacking Nitesh.

Once the enemy has chosen which protagonist it will attack, EnemyAI.Attack() calls EnemyAI.EndAttack(victim). This function then starts the coroutine BattleController.FireProjectile(attacker, victim).

GAME LOOP: LATE STAGE

The late stage of the game loop starts at BattleController.FireProjectile(BattleStats attacker, BattleStats victim, bool critical = false), regardless of whether the attacker was a protagonist or an antagonist. This stage focuses on attacking, dealing damage, and potential defeat.

Attacking

Starting at BattleController.FireProjectile(attacker, victim, critical), a projectile is created and fired across the screen from the attacking character to their victim. This may be subject to change for attacks that make less sense as a projectile, but as of now, that's what all attacks look like. When they are done, the projectile is destroyed and the victim's BattleStats.GetAttacked(BattleStats attacker, bool critical) is called.

Taking Damage

When a belligerent gets attacked, their BattleStats.GetAttacked(attacker, critical) is called. This function subtracts the damage taken from the victim's health. The formula for damage taken is as follows:

> Victim's Health -> Victim's Health - Damage,
> Damage = Attacker's Attack * (1 - Victim's Defense)

Where defense is between 0 (inclusive) and 1 (exclusive). If critical is true, then the following is true:

> Victim's Health -> Victim's Health - Damage,
> Damage = ***2*** ******* Attacker's Attack * (1 - Victim's Defense)

If the resulting value for the victim's health is less than or equal to zero, then the victim is defeated. Otherwise, HealthBar.AdjustHealth() is called for the belligerent's respective

health bar (BattleStats.healthBar), and AdvanceTurn() is called from BattleController.FireProjectile(attacker, victim, critical).

Defeat

If a belligerent is defeated, BattleStats.GetAttacked(attacker, critical) will call BattleStats.Defeat(). BattleStats.Defeat() removes the defeated belligerent from the turn order and deletes the game object. More will likely happen later, like animations, but for now that's all. After a defeat, BattleController.AdvanceTurn() will be called as normal.

In the beginning of BattleController.AdvanceTurn(), BattleController.BattleOver() is called, which returns true if an entire team has been eliminated or false if there is both at least one protagonist and at least one antagonist present. If true is returned, BattleController.AdvanceTurn() is aborted and BattleController.EndBattle() is called. This function does nothing currently, but signifies the end of the battle.

MINIGAME CREATION GUIDE

If you want to make your own minigame, there are a few guidelines that must be followed. If anything is confusing, refer to the Test Minigame scene for examples of a fully-formed and operational minigame in its most basic state.

Starting Guide

To begin, make a duplicate of the scene "_Template Minigame," located in Assets -> Scenes -> Minigames. This will include a few of the items listed below, so you don't have to create them yourself, including the camera, minigame UI scaler prefab, and minigame autonomy prefab. This means you shouldn't need to mess with those too much, just be aware that all UI objects must be children of the prefab titled "Scale With Minigame Window" under the canvas, with the Minigame Tutorial prefab being the last item in the hierarchy of children of the Scale With Minigame Window object. World objects don't have to be the children of anything, and can exist loose in the scene.

Unique Minigame Manager and Key Design Features

Every minigame needs a class that functions as the manager and communicates with the base game. This must inherit from the abstract Minigame class, and use four key variables/functions, as outlined in the section "Running a Minigame."

Camera Requirements

The scene should have one Main Camera, which is included by default, but a couple settings are necessary, or else the screen won't display the minigame in the battle correctly.

- Under "Culling Mask," select only layers with the word "Minigame" in them, and select all those layers.
- Under "Target Texture," select the Render Texture called Minigame View. It should be located in Assets -> Textures -> Minigame View.renderTexture.
- Remove the Audio Listener component from the camera gameObject.

Avoid Interactable Unity UI

Because we have to use a different system from the unity default for tracking cursor positions in minigames, using unity UI that interacts with cursor positions, like the Button type, won't work here. You can use any other kind of UI, just not ones that automatically track the cursor position like Button or Slider. To have UI that does change based on mouse position (e.g. an Image type that moves in relation to the cursor), see the section below titled Cursor Interaction.

Layer Requirements

Every object in every minigame must have its layer set to a layer with "Minigame" in the title, or else they will be invisible in the battle. There are currently four minigame layers (differentiated based on which layers can collide with which, with one for UI specifically), and if you really need a fifth, please let the chat know so that we can make sure that change is translated across the different scenes in the game.

Minigame UI Scaler Prefab

In the prefabs folder is a prefab called Minigame UI Scaler. It's a necessary prefab that scales the UI so that they always fit within the minigame window when presented in game. To use it, add it into your minigame scene, make it a child of the canvas, and make sure all UI are children of the Minigame UI Scaler prefab.

Minigame Autonomy Prefab

In the prefabs folder is a prefab called the Minigame Autonomy Prefab, which is not necessary, but very helpful for testing minigames quickly. Add it to a minigame scene, and the minigame will run on its own, and you won't have to switch to the battle scene every time you want to test your minigame. Also, there is no need to remove the prefab when you do run the battle scene, as the prefab destroys itself automatically if run through the battle scene, and thus there is no risk of including it in a minigame scene.

Cursor Interaction

Because of the way minigames are run, the position of the cursor in minigames has to be retrieved in a different way.

To get the cursor position in world space, use:

```
Vector2 cursorPosition = Lunarwood.MinigameCursor.WorldPosition();
```

To get the cursor position in screen space, use:

```
Vector2 cursorPosition = Lunarwood.MinigameCursor.ScreenPosition();
```

ENEMY AI STRATEGY CREATION GUIDE

If you want to create a new enemy AI strategy, as seen in the EnemyAI script, here are some guidelines that must be followed.

Update EnemyStrategy Enum

Make sure the name of your new strategy is included in the enum EnemyStrategy, included near the top of the EnemyAI class. The order is unimportant.

Corresponding Function

You'll need to create a function in EnemyAI.cs that narrows down the list of all possible victims to the ideal victim (or victims if there is a tie) according to that strategy. This function must have a few key features:

- The function must return type void.
- The function must take in no parameters.
- The function must have the exact same name as the corresponding name in the EnemyStrategy Enum
- The function must assume that all the available targets are listed in the List<BattleStats> called potentialTargets, which can be called easily.
- The function must (provided at least one potential target can be eliminated) modify the items in potentialTargets to remove non-ideal targets.
- The function cannot add new targets to potentialTargets
- The function must ALWAYS keep at least one potential victim in potentialTargets.
- The function must only support superlatives for searching (e.g. if you wanted to make a strategy to find the targets with the highest defense, it must only keep the

target (or targets if there is a tie) with the highest defense in potentialTargets, not the second or third highest.

- The function must support the possibility of a tie if that is possible.

Referring to existing functions TargetStrong() and TargetWeak() will likely help you figure out how to structure your function in a viable way.

COLOR CODING IN THIS DOCUMENT

I'm self aware enough to know this might come up.

Color	Instructional Text	Examples
Black	Main text	Given information
Blue	Headings	Main work
Red	Vocab words and important side notes	Tertiary work, example number, and important side notes
Green	BIG headings	Secondary work
Purple	Formulas	Final answers