Documentation for Work Done for Dungeon Royale

By Alexander Caichen

Table of Contents

Weapon Timing

There are three main variables to assign when creating a WeaponItemData ScriptableObject:

1. Fire Rate: the amount of seconds it takes to fire the next shot.
2. Clip Size: the amount of shots one can take with that weapon before it needs to be reloaded. This is set to 0 for weapons that do not need to be reloaded
3. Reload Time: the amount of seconds it takes for a weapon to reload if it does reload at all.

As of time of writing, the code to set up the next time a shot can be fired (variable `nextFire`) should be found under `SetupProjectile()` in CombatEntity.cs.
For how shots are influenced by `nextFire`, go to `Update()` in PlayerController.cs and `Attack()` in EnemyController.cs. (I did not code these functions, just mentioning them here for convenience).

Projectiles

Various added effects on projectiles to make the game more interesting.

Note: this section corresponds to commit `3f31969361ac061b9e14cd9c4c303b710129c3e5` on the "uniqueshots" branch.

Assigning Effects

Status effects and most weapon behaviors are assigned through WeaponItemData files. Status effects come in the form of arrays while behaviors come in variables. For example, Poison (effect) is an array of 2 values: time and intensity, while SuperPierce (behavior) is a Boolean.

When a weapon is fired, the various effects and behaviors stored in the WeaponItemData are packed into arrays for ease of transport through the various functions that are called before a projectile is actually instantiated. The functions `propertyPacking()` and `effectPacking()` located in WeaponItemData.cs help "pack" the projectile behavior and effects from the files into Boolean and Float[] arrays respectively. If these behaviors and effects are not packed, various function in CombatEntity.cs (`RpcFire()` and `LocalSpawnProjectile()`), Projectile.cs (`Activate()`), and ExplodingProjectile.cs (`Activate()`) will require at least 5 more inputs to run, which is very messy.

When information is assigned to an instantiated projectile prefab through `Activate()` in Projectile.cs, the arrays are "unpacked" and their contents assigned the various status effects and projectile behaviors.

Status Effects

CombatEntities contain arrays or variables (depending on the status) to time how long a status lasts and the severity of the status effect for each entity.

```
//Time, %
protected float[] poisoned = new float[]{0, 0};
protected float[] slowed = new float[]{0, 1};
//Time, dmg
protected float[] burnt = new float[]{0, 0};
public float silent = 0;
```

Code for inducing a status effect when a combat entity is hit by an attack is located in `OnTriggerEnter2D()` in CombatEntity.cs.

- Due to AOE attacks currently not initializing status effects/status arrays, the code first makes sure that the length of the status arrays are > 0 (prevent null error messages).
- The code then checks if the gameObject/projectile hitting the player has a time > 0 listed for a status (verify if the projectile does indeed cause status) for each of the statuses.
- If the projectile has a time > 0 for a status:

- Set the 2nd index of the status arrays to a more serious version of that status (Ex: Set `burnt[1]` to 60 if it had previously been 30, so burn now does more damage)
- Either start a status Coroutine (if status is not active before) or update the amount of time the status lasts (Ex: if a bullet that causes slow for 5 seconds hits an enemy that has 1 second of slow left, the enemy now has 5 seconds of slow)

Each status effect has a Coroutine in CombatEntity.cs to control that status's behavior and prevent clutter in CombatEntity.cs's `Update()`. The Coroutines generally make sure a status lasts for the correct amount of time with some variation in between, ending when the time remaining for respective status is <= 0.

Poison
Causes % of MaxHP-based damage every second it lasts. The percent is the 2nd index of its status array.

When taking poison damage, the CombatEntity will briefly flash green. A separate Coroutine (`poisonColoring()`) is used to time/coordinate this green flash alongside Silence coloring. This is so if the player supposedly turns purple due to Silence in the middle of a flash, the player will turn back to purple at the end of a flash instead of being tinted white (normal) if Silence continues. (Also applies to the opposite coloring transformation, where Silence is applied in the middle of a Poison flash).

The current green color for poisoning is stored under Color variable `poisonColor` in CombatEntity.cs if later developers want to change the color.

Burn
Causes a fixed amount of damage every second it lasts. The amount of damage stored in the 2nd index of its status array.

Dashing by players gets rid of the Burn status effect. Code for this can be found under `OnDash()` in PlayerController.cs.

Due to Dashing potentially removing Burn (setting `burnt[0]` to 0) during the `WaitForSeconds(1)` in the Burn Coroutine, an extra IF check is added in the Burn Coroutine to decide if the Burn timer will be decremented. This prevents an extra Burn Coroutine from starting from `OnTriggerEnter()` before the first Coroutine completes, resulting in more than one Burn Coroutine being active at the same time.

Notes for Burn and Poison: There's a condition added in `RpcTakeDamage()` that allows Burn and Poison damage to sidestep defense stats. `ownerName` for Poison and Burn inputted into `RpcTakeDamage()` is their respective status names and `ownerID` is set to -999. There is some code in `Die()` of PlayerController.cs (line 816 at time of writing) that deals with deaths and death-messages caused by status effects (otherwise the player becomes immortal after supposedly dying by burn/poison).

Slow
Decreases movement speed. The 2nd index of its status array is a *multiplier* for speed (Ex: 0.7 causes speed to decrease by 30%).

Slow impacts players in `FixedUpdate()` in PlayerController by multiplying `rb.velocity` by the slow multiplier if Slow is in effect.

Because Enemies rely on `navMeshAgent.speed` instead of `rb.velocity` like for Players, a separate `slowtimer()` Coroutine is implemented in EnemyController.cs that decreases navMesh Speed at the beginning of the timer then restores it at the end.

Silent
Prevents Player from using magic-related actions. It simply uses a Float to record the amount of time silence lasts.

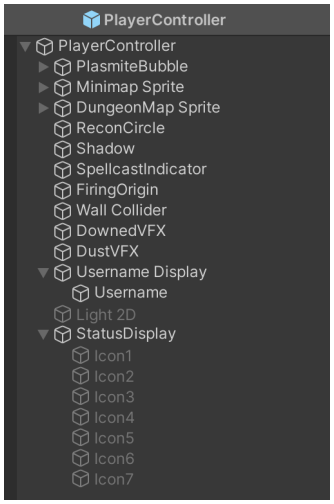
Since Silence is exclusive to Players, a separate `siltimer()` Coroutine is implemented in PlayerController.cs that makes the Player tinted purple and the status icon pop up. The `silenceColor` variable in CombatEntity.cs stores the tint color for Silent. Variables related to Silent are stored in CombatEntity.cs despite not influencing EnemyEntities in case developers decide to make Silent affect Enemies in the future and so all status-related variables are located in one place, easing access.

To prevent the end of the Silence timer from changing Player tint to normal in the middle of Poison's green flash, the variable `poisonColored` set under `poisonColoring()` is used to check if a flash is currently occurring or not.

There is a Silent checker under `OnDash()` in PlayerController.cs to prevent dashing when Silent and another under `OnSlotX()` to prevent spell usage when `silent` > 0. There is also a check under `FireWeapon()` in WeaponHandler.cs to prevent magic weapons from firing when `silent` > 0.

Status Display

There is a Horizontal Layout Group called StatusDisplay under all CombatEntity Prefabs shown here:



Nested under StatusDisplay are as many images as total status effects implemented. As of time of writing there are 7 status effects so there are 7 images. Layout Groups allow one to easily center images nested under it based on how many of the images are Active (the little checkbox to the left of the name of a gameObject shows if the Object is active or not).

Examples:

 <- 2/7 Active

 <- 5/7 Active

This Layout Group is referenced in the CombatEntity.cs code as a public Gameobject called `statusDisplay`.

There is a (Linked)List of Sprites in CombatEntity.cs called `statusList` which helps order the status icons to be displayed in the Layout Group (oldest status on the left, newest on the right). There is also an array of Sprites under GameManager.cs called `statusIcons` containing all sprites of current status effects.

- Remember to update this list in the Scene when new statuses are added

Whenever a Coroutine for a status effect starts, the Coroutine would add that status's respective icon from `statusIcons` to `statusList` then call `updateStatusBar()` in CombatEntity.cs.

`updateStatusBar()` iterates through the images nested under `statusDisplay`, changes them into their respective image from `statusList` (so the first nested image becomes the first Sprite in `statusList`), and activates the GameObjects those images. If not all statuses are activated (Ex: only affected by 1 status), the rest of the GameObjects (6) will be deactivated.

The GameObject list is very short (again, currently 7 elements long) and is unlikely to increase to a long length such as with 100 elements so there should be no potential speed/efficiency issues caused by constant list iteration if an Entity is constantly bombarded with status effect-inflicting shots.

Just before the status Coroutine ends, the status icon is removed from `statusList` and `updateStatusBar()` is run again.

Behavior

Different bullet behaviors or shooting patterns.

Superpierce:
Bullets pierce through walls/obstacles.

There is a condition added to the first IF statement under `OnTriggerEnter2D()` in Projectile.cs that prevents the projectile from self-destructing when touching a "NonWalkable" object when `Superpierce` == True.

Multishot:
Fire multiple shots at once. Clip size is still used up according to how many shots are fired, not how many times players click to fire.

Code related to this feature is located in WeaponHandler.cs. `FireWeapon()` determines what weapons utilize this feature depending on the weapon's name.

`SpreadFire()` manages how the extra bullets are fired (`extraBullets` signifies the bullets fired other than the original bullet fired towards one's cursor, so 3 `extraBullets` means 4 bullets are fired in total). Extra bullets are fired in a left-right order: meaning the 1st extra will be fired (`angleDiff`) to the left of the original bullet, the 2nd fired (`angleDiff`) to the right of the original, the 3rd fired (`angleDiff`) to the left of the 1st extra, the 4th fired (`angleDiff`) to the right of the 2nd extra, etc.

Bounce:
Bullets bounce off inanimate objects (walls and obstacles).

Bounce is achieved by modifying the PhysicsMaterial2D (or "Material") in the Collider2D of a projectile to have a Bounciness value of 1. Since directly modifying Bounciness gives a *"unity Property or indexer 'Collider2D.bounciness' cannot be assigned to -- it is read only"* error, a new PhysicsMaterial2D object with a Bounciness of 1 was created in-code and assigned directly to the material of the projectile's Collider2D.

In addition, because Bouncing uses OnCollisionEnter2D, `isTrigger` must be turned off (False). Bouncy projectiles would not be destroyed by walls anyways now so disabling OnTriggerEnter for these projectiles does not matter.

The `Activate()` of Projectile.cs accomplishes the above when the WeaponItemData which a projectile is based off of contains a Bounce value of True.

Before using PhysicsMaterials I tried several attempts at manually coding-in bouncing behavior under `OnCollisionEnter2D()` of Projectile.cs by changing projectile direction after collision with a wall, but these attempts always ended up with inconsistent bouncing angles or projectiles occasionally piercing through walls despite Superpierce not being on. Using PhysicsMaterials provides the most consistent bounce angles and does not cause projectiles to go through walls.

There are two existing bugs so far:

- Bounced projectiles spin (not very noticeable considering that the only projectile that bounces so far is in the shape of a circle from the Moon Wand).
- Projectiles do not bounce off the sides of chests.

Boss Behavior

Boss stats and data related to its behavior can be found in BossData ScriptableObject.
Boss behavior is based on “phases” and “subphases”.

- Subphase: a behavior that occurs for a certain amount of time (Duration within the Subphase)
    - Each Subphase includes its own WeaponItemData, Attack, Defense, Speed, etc. variables to allow for greater flexibility when programming Boss behavior when using BossData.
    - If the Phase doesn’t change, a Boss will loop through the current set of Subphases in the current Phase over and over in order.
    - Ex: the boss starts with a subphase where it stays still and fires shots quickly (ROF is controlled in WeaponItemData). After 4 seconds it switches to a subphase where it chases the player. 5 seconds later it goes back to standing still.
- Phase: a set of subphases that a Boss uses when reaching a certain HP (`SwitchHP`)
    - There is at least one Subphase under each Phase.

The “Notes” variable in BossData allows developers to add developmental notes about that Phase/Subphase within the Unity editor. It does not influence boss behavior.

Code to control Boss behavior can be found in BossController.cs:

- `Update()` mostly controls Subphases/Phases switching. General behavior (such as roaming) is still controlled by EnemyController.cs’s `Update()`.
- `UpdateStats()` updates the Boss’s stats to the stats recorded in corresponding Subphase in the boss’s ScriptableObject.

If you want to add a boss to the game to be spawned, please add the corresponding BossData ScriptableObject to the MasterBossList ScriptableObject then attach the BossData to the GameManager in the MainDungeon Scene.

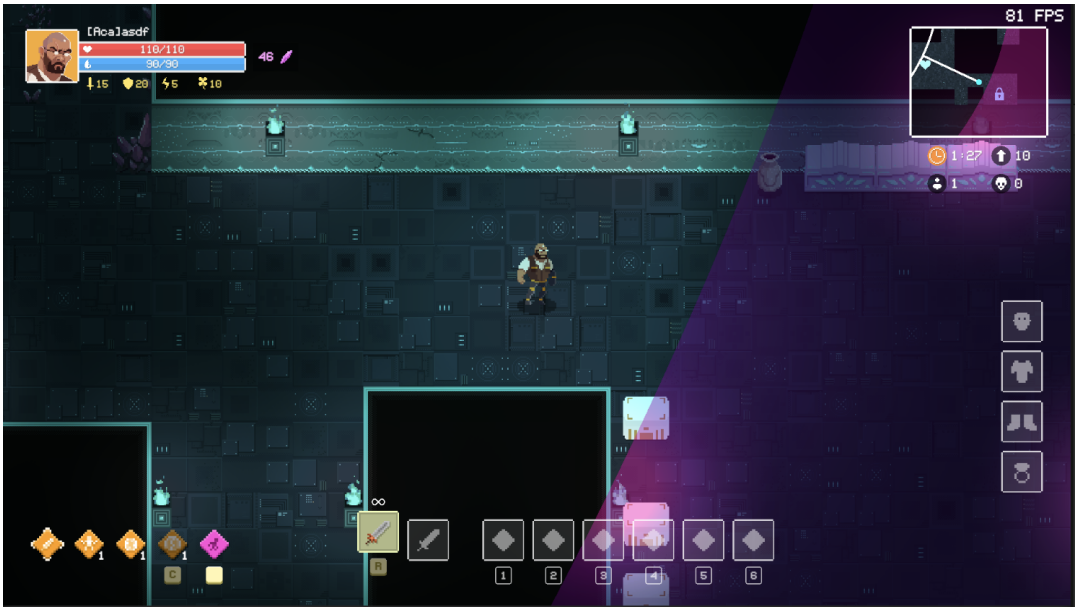
Other than having BossController.cs attached instead of EnemyController.cs, Boss prefabs currently contain no differences from normal Enemy prefabs.

Minimap

Implementation is based on https://blog.theknightsofunity.com/implementing-minimap-unity/ and https://www.youtube.com/watch?v=28JTTXqMvOU&ab_channel=Brackeys.

Storm

A circle that surrounds the map at the start of the game. It slowly shrinks at regular intervals around random centers (so at the end it does not always shrink to the dead center of the map). Players outside of the circle will be damaged by a certain amount every second, and the amount of damage increases over time. Enemies/Monsters are not damaged from being outside the circle.



^Bruno next to the edge of the storm.

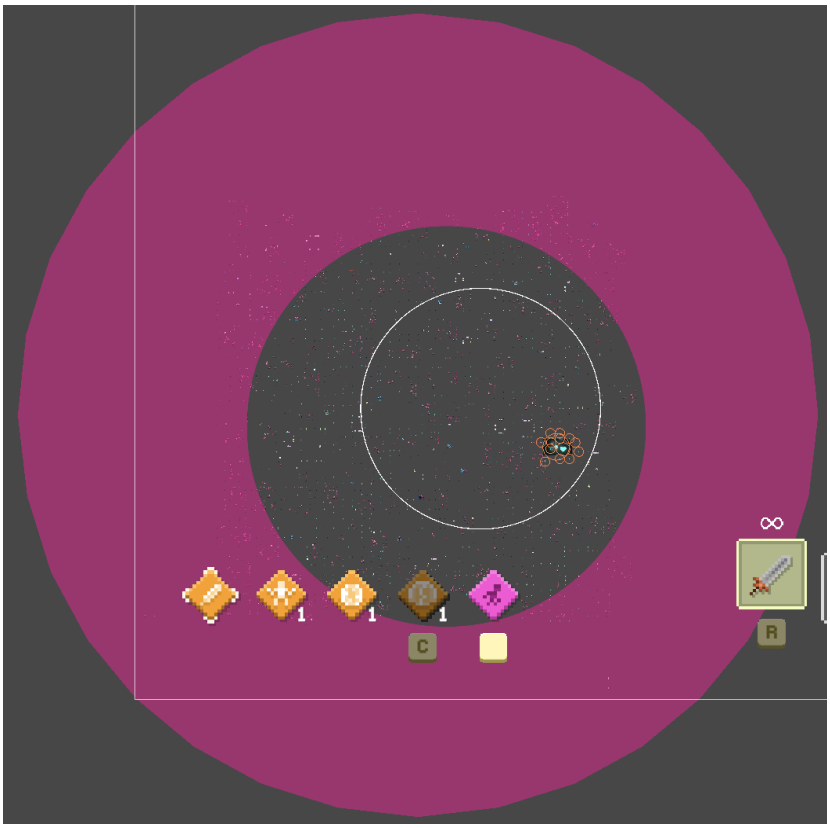
Setup

A Storm script is attached to the GameManager script in the MainDungeon scene. At the start of a game, the GameManager will call `Init()` from Storm.cs to initialize a Storm object (see `SetUpGame()` in GameManager.cs).

The storm itself is actually a flattened sphere rather than an actual circle. The Storm finds the center of the dungeon, sets the coordinate as the sphere center, then uses this data to set up the size of the sphere so that it completely covers the dungeon.

```
x = y = (float) gameManager.dungeonGenerator.SizeGetter()/2;
//starting radius
radius = Mathf.Sqrt((x*x) + (y*y));
```

A custom shader known as SphereMask.shader is used to control the behavior of the visuals of the sphere. As shown below, the shader makes the storm have two "layers": one tinted and one "empty" (directly within the tinted region, the inner circle). The pink portion is the area that causes damage.



Within Sphere.cs:

- `_GLOBALMaskPosition`: determines center of inner circle
- `_GLOBALMaskRadius`: determines radius of inner circle
- `_GLOBALMaskSoftness`: set negative to make inner circle "empty"

Originally the tint color for the space between the inner and outer circles was set directly in `Init()` of Storm.cs (line 69 as of time of writing). I believe the color is now set in `UpdateGameInfo()` of UIManager.cs (a UIManager script is attached to GameManager in the MainDungeon scene).

Behavior

Storm behavior is mostly located under `Update()` and `shrinking()` in Storm.cs.
Four variables influence the shrinking cycle of the storm:

1. `waitTime`: Amount of seconds before the inner circle starts shrinking.
2. `shrinkTime`: Amount of seconds shrinking lasts once started.
3. `timer`: Times the waiting/shrinking process. It is set to `waitTime` or `shrinkTime` then counts down to 0. When it reaches 0 it is time to switch to the other process.
4. `shrink`: True if the circle should be shrinking, false if not. In conjunction with `timer` helps switch from one process to another.

The `shrinking()` coroutine works by calculating the radius and center of a new inner circle (`nextCircle`, initially created in `Init()`, shown in the previous picture as the white, drawn circle within the inner circle) and slowly modifying the actual inner circle (the border between the pink and untinted region) into the size and position of the new circle while `shrink` is True. This process should last the same amount of time as `shrinkTime`. Afterwards, a new `nextCircle` is generated for the next rendition of `shrinking()` to use.

- The reason a Coroutine is used instead of putting everything in `Update()` is because we do not want to clutter up `Update()` too much and because we do not want to go through 4 different if statements every millisecond just to shrink the inner circle a bit, which also seems inefficient.

The circle will stop shrinking when the inner radius is <= 1 as we do not want the radius to become negative (see line 88 in Storm.cs as of time of writing).

For questions about the line drawing function (`UpdateToCircleLine()`) used to draw the line in the minimap, please ask Robert.

Damage

Whether or not the Player is outside the circle or not is calculated using `StormDistance()` in PlayerController.cs. This function simply subtracts the player's distance to the inner circle's center from the current radius of the inner circle. Dealing damage to the Player via the Storm if the player is indeed outside the circle (positive output from `StormDistance()`) is done under `Update()` in PlayerController.cs. (Search for "StormDistance" in the file for the location of the function. There is only one time this function is used).