Beri Gamedev Journal

To Start:

We will start this journey by making a few things clear:

!) We will NOT get distracted from things that aren't important to the main game

!!) The main battle system functioning well is our main focus. Things that follow as consequences of our prototype we will discuss later.

!!!) The game's draw will be the battle system.

- We will implement the items, characters, subsystems, etc. later.
- Ber: beats the shit out of a demon. That's the hook.

Overview

Development progress will cover these systems:

Battle System:

- Singletons
- State machine
- AI
- Animations

- Each of these components will inevitably have subsystems to consider. Don't fret, approach them calmly and plan them appropriately

Overworld:

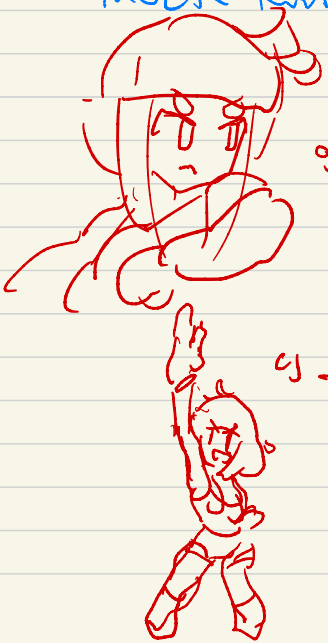
- Item storage and collecting
- Item and equipment management
- Movement

Refinement:

- Animations
- Music
- More characters
- etc.

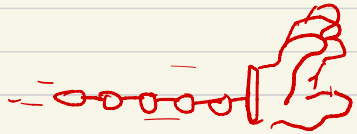
Gameplay

- Battles are turn-based with each action requiring some form of command to perform.
  - Grabs, attacks, magic, everything is associated with an input command.
  - Alone, attacks deal merit damage, but when combined with other moves, attacks become much more potent.



GT | AT

a) Bori selects grab upthrow, and up attack as her move.



b) Time the grab input

c) Times the "up" input.

d) Times the up and attack input, so on and so forth.

Battle System: Approach

- State Machine:

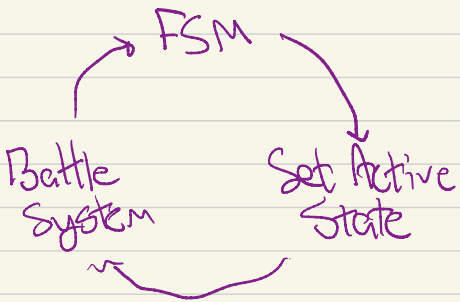
  The state machine will track which state of the game is current from a finite list of states:

FSM
→ has
→ States
→ which dictate
→ Behaviour.

FSM:
- → Player Turn
- Enemy Turn
- Win
- ⋮

State:
When active:
Communicate with the battle system to display graphics, change stats, etc.

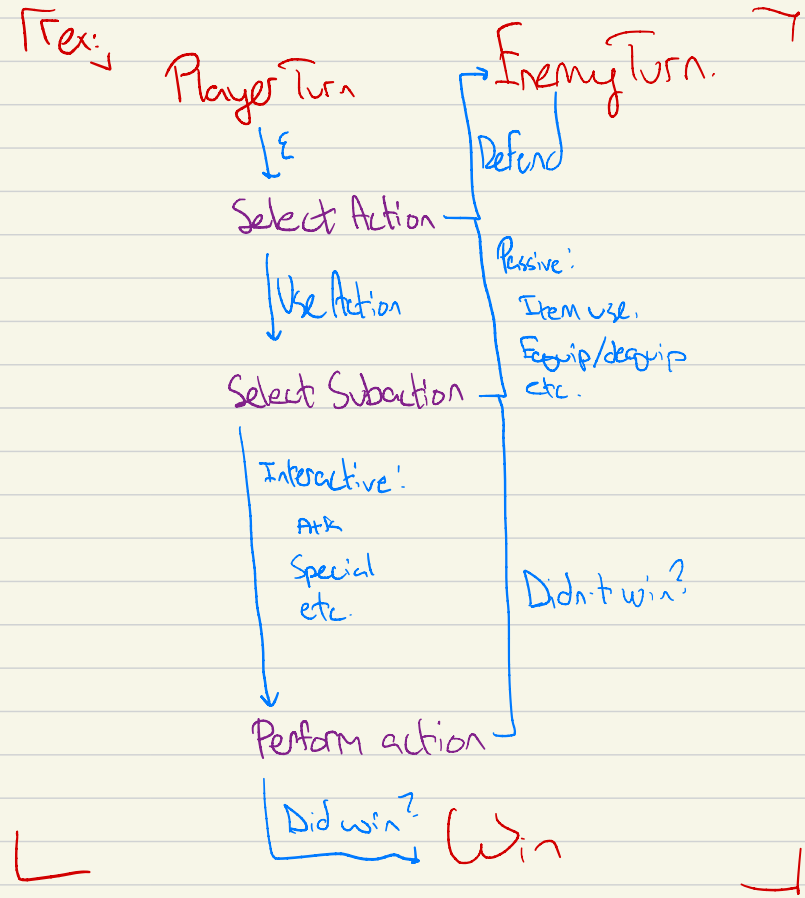
Battle System:
Given a state:
- Give or take control
- Run enemy AI
- etc
- Select next state depending on outcome.



Battle System: FSM

The FSM provides the set of states reachable from any other state. A state may have a set of substates that are only reachable from that state.

These substates should pertain to actions and events that only concern the data and entities of that state.



State Communication with Battle System

Overview:

Each state simply acts as the "go ahead" for the Battle System to enable and disable certain events, such as what GUI to draw or what animations to play.

When a state is active:

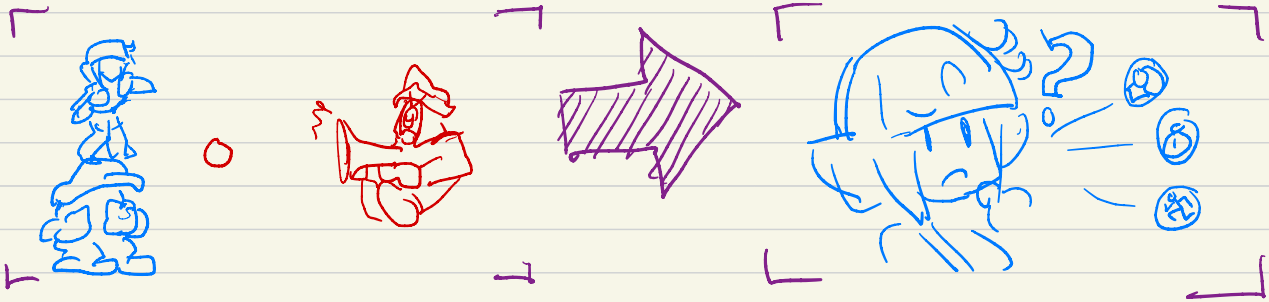
- Swap the current active state in the FSM
- Notify the Battle Manager of the state change
- Cache the state and it's behaviour into the BM, Draw appropriate GUI, etc.

FSM holds active state and alerts BM of changes → State contains some default methods and holds appropriate behaviour → BM manages entities, values, current state, and any behaviour not state specific.

BM Responsibilities

Overview:

FSM sets current state to PlayersTurn. BM communicates to the player entities that they can now move, select actions, etc.



Here, the current state is EnemyTurn. The BM selects the enemy who will attack, provoke the enemy's generic Attack method. The enemy's AI selects a move, and the player can respond.

Then, when the turn is over, the state becomes PlayersTurn. The BM selects the active player character and tells them to start drawing the GUI cached in the state class

System Overview

FSM

Holds the current state and signals BM of state changes

Battle Manager

Holds data pertaining to all UI, entities, stats, etc. Each state dictates what is done with the data.

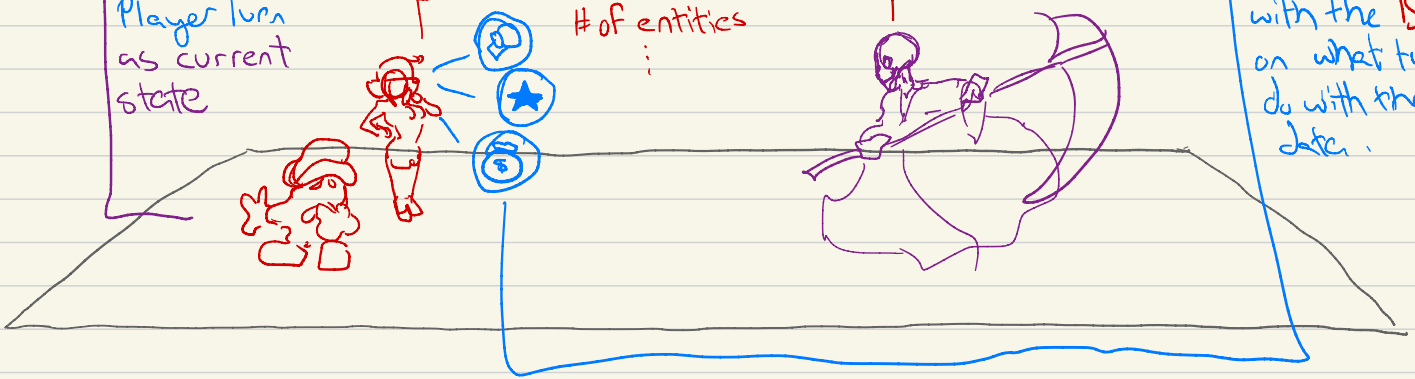
States

Contains methods for drawing appropriate UI and what tasks to enable/disable.

FSM has Player Turn as current state

Stats
Turn order
# of entities
⋮

State communicates with the BM on what to do with the data.



Implementation

States store functionality
Battle Manager stores every state, and data pertaining to units on the battlefield.

- Each state is a unique class that inherets from a base abstract class.
  - Class Methods:
    - Activate();
    - Deactivate();

/* When activated, we may utilize any functionality we want. */

Statemachine Update

The previous implementation was ok, but we need to expand. The issue is there wasn't enough modularity in the base class. As such, searching for specific UI elements in a parent class was unfeasible.

StateManager:

Contains:

List of UI objects characters, character data, and any other important information.

Methods:

Any methods that are important for the given context

States:

Contains:

Methods for activation, deactivation, and any others necessary

Responsible for:

Dictating flow of the game, enabling or disabling UI, controls, etc.

Overworld:

Contains OW methods + objects

Battle

Contains Battle methods + objects

Etc...

Whatever else is needed

OW: Walking

OW: Menus

BTL: Player

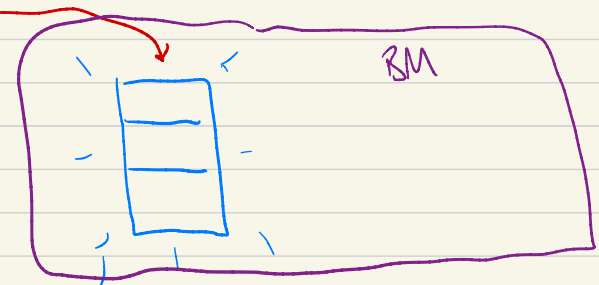
...

Data Manipulation

Each state get's a reference to the BattleManager. The state can then choose which elements in the UI to render, and those elements can have classes of their own.

State:

- bm.uiSample.SetActive(true);



For now, we'll worry about basic character and enemy interactions.

UI:

click button, spawn submenu, etc...

What Should Battle Entities Have?

A battle entity needs to be compatible with the player characters and the AI controlled enemies.

We also want to reduce the risk of pollution by having too many scripts accomplishing too little.

Player Battle Entity

- Must gather data from player's adventure thus far
- Must be able to differentiate between player and partner.

Enemy Battle Entity

- Have limited moves, items, etc.
- Must be able to get it's information from world.

These two types must be able to take and recieve damage, and modify stats through the current state. A parent class "Battle Entity" can accomplish this.

Checkpoint:

8·25·22

What's Done so Far:

- Functioning State Machine with states affecting the data of the Battle Manager
- Battle Manager UI elements which can be affected by the states.
- Core system functionality established.

What Needs to be Done:

- Basic communication between entities.
- A UI element that can spawn more UI elements [for the player turn]
- Set up basic UI and gameplay controls.

Communication to Battle Entities

We want a version of the BM to be stored in any overworld scene so we can send data to it regarding what entities to spawn and how many.

To begin, we need to discuss what battle entities are and how to best utilize them.

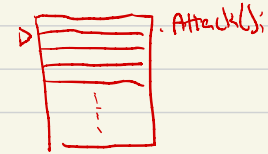
An "Entity" is any character in the game that has stats specific to it. The player is an entity, so is an enemy, a boss, partner, etc.

Entities. When they collide on the overworld...

... Send a message to the BM, which includes:

- What entities to spawn.
- Where to spawn

Store the entity data in the BM to use for later.



Initializing Field

When we encounter an enemy, or whichever event may cause a battle, assign a list of characters to the bm via reference [this can be done via gamemanager]

Field.

Basic Setup

We will build a system with a few basic states:

- Player Turn
- Enemy Turn
- Win / Loss

For sake of simplicity, we will make a simple game with these 4 states.

To build the scene:

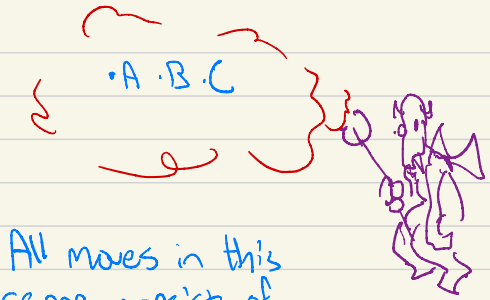
1) Send entity data to BM
2) BM spawns in entities
3) BM Determines turn order, represents order as a list.
4) When player turn:
   - Draw GUI to perform attack, defend, or use item, etc.
   - Set controls to specific set for the scenario.
     i.e.) Remove directional input on enemy turn.

Moves

Moves are any action a player or enemy can perform. They contain data and methods on how the move should behave [i.e., what animations to play, how much damage should be dealt, healed, etc.

Each move takes data from character objects.

Bori and co. will be slightly different, as she can create moves on the fly.



All moves in this game consist of
- Animations
- Hitboxes
- Behaviour.

Behaviour: "Shoot gun at enemy"

AI needs
Movelist
needs
Move

Moves [cont.]

Beri has modules which build a compound move that can be performed. Each module can combo into one another and create unique behavior.

Beri, depending on the move will need to position herself either up close to the enemy, or allow her to stay still.

So, the first move should govern which.

Module Types:

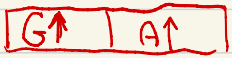
Solo:
Modules performed by one actor

Handoffs:
Relinquish control of one actor to let partner take control.

Press:
Tap button

Hold:
Hold then release button

Beri:



"Grab enemy, throw up, then uppercut"

Beri and Partner:

Beri grabs opponent, throws forward, hand control to partner.

Partner performs their own attack

Modular Move Implementation:

Modular moves have similar properties to standard moves, with the exception of inputs being added.

Modules have their own associated
- Animations
- Projectiles
- Behaviour

Move Lifecycle:
- If player:
  - Load Module. Each module is a small move with a hitbox, animation, and behaviour.

ex

G↑ — Grab has it's own animations, the hitbox is a projectile, which the behaviour says "pull back opponent"

A↑ — Hitbox is stationary, which behavior says "punch opponent up"

Move Implementation:

PLAYER

Battle Manager

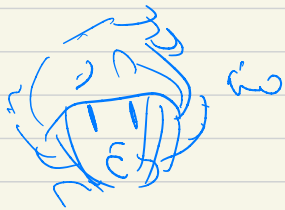
- PlayerTurn state is active, player can select moves stored in the entity's character

Entity

To perform basic attacks, choose the modules you wish to do. Store them in a list to execute one at a time.

Move

Run this current move to success or fail. When complete, if success, move to next move. If not, fail and end turn.



Move order

[.....]

execute first move.

Besi selects two moves, grab-uppnrow, uppercut

Move Implementation [cont.]

Player

" Set up moves, hit execute

a) • "execute" will call the player turn class method, given the list of moves, the character, and the animations.

• This method calls the individual character's move "execute" functions, which have three states.

- OnStart : Play animation, launch hitboxes, etc.
- OnSuccess : Notify BM that the turn was a hit.
- OnFail : Notify the opposite, play a specific fail animation.

Checkpoint [9.20.22]

What's been done?:

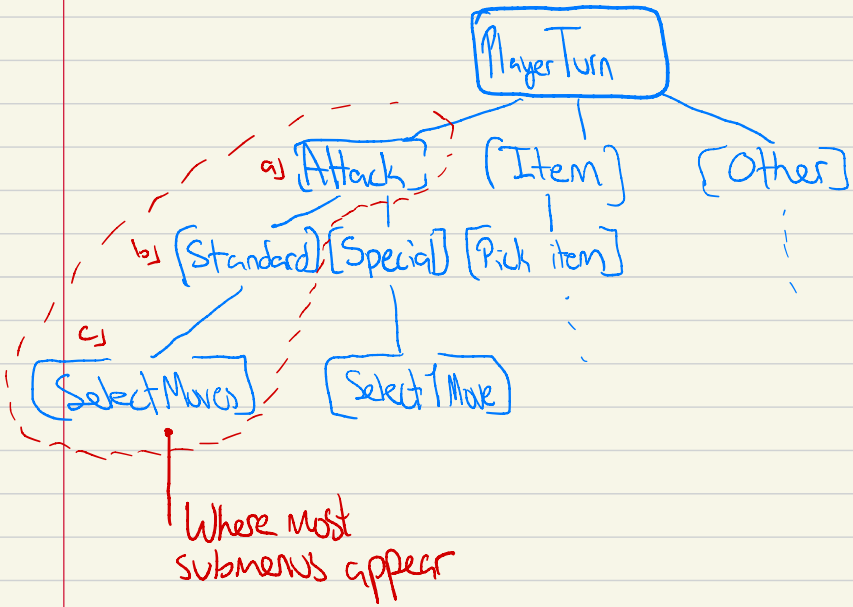
- Boilerplate for BM
- Basic Communication between the BM, entities, and other entities

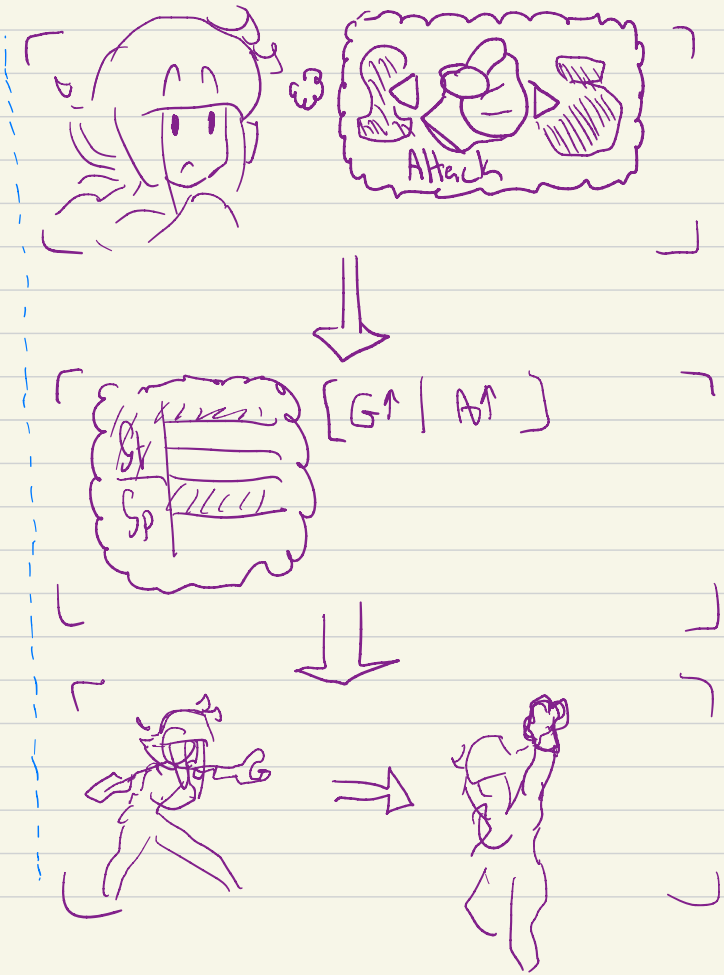
What is to be done?:

- Basic battle, with modular attacks.
- Basic enemy AI
- More UI, states
  - Enhance player UI by showing modular menus

Player Turn

- The flow of the player turn looks something like this:

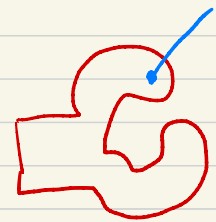




Entity Interactivity

- For our moves to work, we need an easy way to communicate directly with entities.
- We can accomplish this through the battle manager, creating methods to manipulate actor animations, health, etc.

This implies every object in the battlefield needs a reference to the BM.



Move Structure

Moves need to communicate to the caster what exactly they are doing when they are cast.

We have our obvious:

```
Initiate();
onSuccess();
onFailure();
```

... but how do we make these modular enough to support a ton of moves?

Moves come in 2 unique types as of now:

- Grab: Bring opponent close
- Physical: Go up and attack

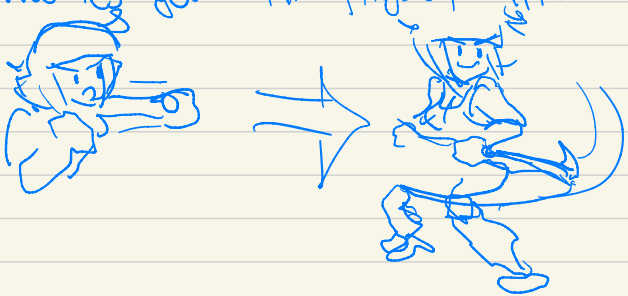
Depending on what weapon Bri is wielding, the move changes.

2 types of weapons for now:

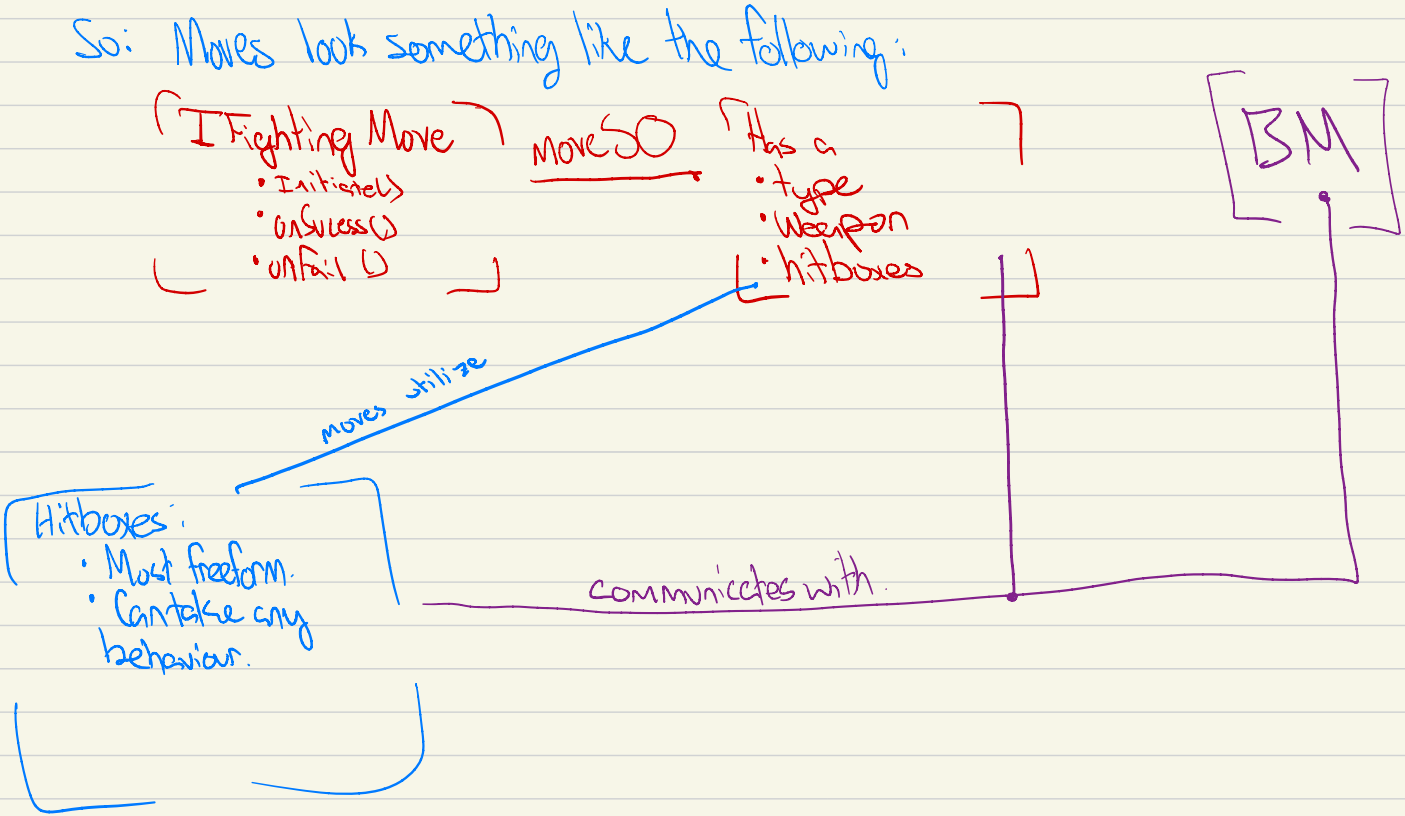
Bare Fist:
- You get Grabs

Scythe:
You get a projectile

Each weapon also modifies your main physical attacks.



So: Moves look something like the following:



Starting a Player Turn

- The player has stamina. Each move the player wants to use takes up one stamina.
- The Move order will be represented as a queue.

| Move 1 | Move 2 | ..... N |

First move to be executed.

When a move is called:

- Notify the BM when the move is initiated, successful, or a failure.
- The BM will handle the underlying logic, as well as assign the appropriate dependencies.

For now, the move list will be hard coded in.

Beginning a Battle:

Now the real work begins. The BM needs to be tested.

- Upon starting:
  - Take all info about battle entities in.
  - Spawn entities.
    - Two main types:

Player	Enemy
• Player Controlled	• AI Controlled.
• Move list tied to save file	• Move list tied to specific entity.

Both have moves

Both have animations that talks to the BM

This means we need a means of reading what moves the player has from a file, and finding the asset in the folder.

Setup

1.

Overworld Enemy:
- Contains a list of enemies for the encounter
- Each enemy has
  - Movelist — What we've discussed
  - AI — Enemy entity will need a specific method to call their AI.
  - Hurtbox
  - Sprites/Animation

Overworld Player:
- Contains a movelist that is continuously updated as player progresses
- Contains a player/partner specific entity which is what will be used in battles

2.

BM { Player | Enemies | Methods to their behaviour, animations, etc. | ... }

Upon setup:
- Set turnorder
- Spawn Entities
- Initialize Variables
- etc...

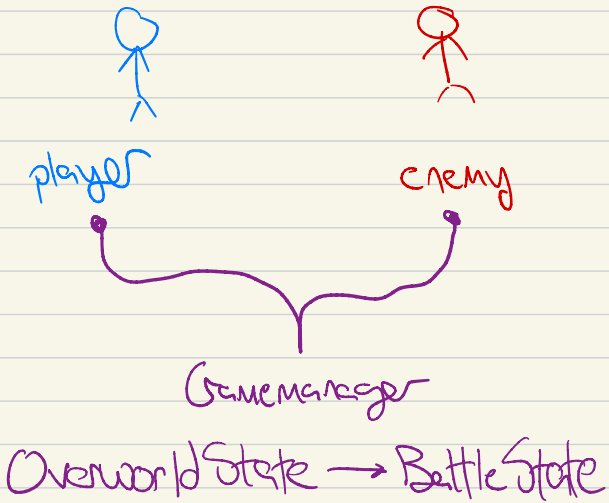
While Running
- Draw UI
- Manage States
- Update Data
- etc...

The BM's responsibility from this point out is to be the main interactive element in the scene. All necessary data to perform a battle is passed through here.

Setup [cont.]

Upon collision with an enemy the main GameManager should be updated with the information of the of the collided enemy.

Overworld Characters that get sent are those related to the event that caused the battle to start.



OverworldState → BattleState

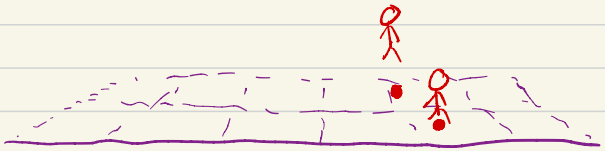
Setup [cont.]

The playfield will have spawn positions for each entity loaded in.

a.)

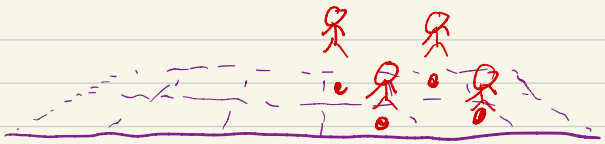
b.)

c.)



d.)

e.)



End of Personal Notes | 11-8-22