DELIVERABLE 1

Executive Summary

Title: Modular Status Effect Management and Combination System

Member Roles:

- Risa: Github lead, Programming
- Jackson: Documentation, Programming, Art
- Maya: Documentation, Art, Programming

Overview:

- Our custom module will help mitigate rewriting code and make it easier to apply these status effects in a modular and effective manner. Ultimately, the intent of this module is to make designing a game with this type of system easier in a way that doesn't require designers to know intimate details about the inner workings of the status effect system. Since systems like status effects exist in such a wide range of games, there is value in making this easier to do for designers, and less tedious to implement for game developers. Creating a reusable system that improves the human performance of making a game, and the computational performance in processing interactions through running this in a C++ module, is a valuable thing.
- Our game will be a side scrolling platformer that has a player and enemies that can apply various status effects to each other through a combat system or through environmental effects. This game will implement themes of magic and elemental effects for displaying these effects in both a practical and visual way that is likely to be familiar to players.

Challenges Anticipated:

- Due to none of us ever implementing a custom module in Godot before, it could be hard to decide on a module with the proper scope of what we are able to do and what is within our capabilities.
- Inexperience with custom resources, and managing the data in that way might cause issues when trying to collect and use Recipe data within the module for effect interactions, could lead to implementation problems during development.

Module Specification

New Classes and Resources:

- The module will comprise of two core classes that are added as nodes within the Godot editor:
    - Entity (Inherits CharacterBody2D): This class will be applied to anything that needs to interact with the status effect system; like a player, a companion, or an enemy.

      The purpose of this class is to hold information about the Entity's health, max health, defense, resistance, speed, damage, friction, and possibly additional properties.

      These properties will have a base stat, a current multiplier, and a current offset in order to apply changes most effectively and allow for efficient interaction between the status effect system and the entities.

      These properties need to exist in this base class because StatusEffect nodes that modify features of an Entity need a universal class to interface through. So, while the actual Entity, whether the player, an enemy, or otherwise, may not be known, the status effect system will work on them all regardless.

      This Entity class can then be expanded into more specific classes, like a dedicated enemy or player class.
    - StatusEffect (Inherits Node): This class will be responsible for holding onto modifier information, or other effects that might need to be applied to an Entity.

      This modifier information could include damage over time, movement effects, a change in maximum health, a change in defense, a change in damage dealt, or other modifications.

      The StatusEffect node, once customized with its properties, can be applied as a child to an entity class and then can do its modifications on the parent Entity class without that parent needing to worry about applying the changes itself.

      The Entity class will have properties that hold onto multiplier or offset information for StatusEffect nodes to modify. For example, it might have its base speed, but then it would also have a current speed multiplier property that is added to or subtracted from by the StatusEffect node and then accounted for within the Entity's own movement script. This would be a way of accumulating different StatusEffect node changes together into one calculation instead of trying to do those changes directly through each StatusEffect node or iterating through StatusEffect nodes in each Entity, creating potential performance overhead. For other changes, like those made to their current health through something like a poison effect, this could be directly monitored and applied by each individual

StatusEffect node in their scripts.

    This would manifest through the ready function of the StatusEffect node applying its corresponding multipliers and offsets to its parent's properties. Then, the exit function would reverse those same effects. The StatusEffect node would also have its own physics process function that can apply a certain amount of damage or other instantaneous changes every certain amount of physics ticks.
- There will also be a new type of resource called a Recipe:
    - This Recipe resource will be used to define interactions between StatusEffect nodes and the resulting StatusEffect node from that combination.

      This resource will contain information regarding the input StatusEffect nodes, the type of interaction, and the resulting StatusEffect node. For example, it could define the inputs as "fire" and "water" and could say that when these two mix, we will remove the fire StatusEffect node, resulting in no additional statuses. We could also define something like "oily" and "fire" as the inputs, and say that those two will have a combining interaction that consumes those two inputs and creates a new stronger status called "inferno", taking the longer duration of the two.
- There will be a singleton, called an Autoload script in Godot, that will be the manager of StatusEffect interactions:
    - This singleton is responsible for getting signaled when new StatusEffect nodes are added to an Entity, going into that Entity's information and checking that new StatusEffect against previously existing ones for interactions described in the Recipe resources.

Technical Architecture and Design Patterns:

- Composite design pattern: This design pattern best fits our implementation because it compliments the normal node structure of Godot, and allows for separated responsibilities between StatusEffect nodes without trying to combine them all into a larger system.

  This system works by adding the StatusEffect nodes as children and using them to compose the Entity and those StatusEffect nodes can then modify their parent properties and do other over time effects. This also makes it so that the Entity itself does not need to know that the StatusEffect nodes exist for them to make changes, overall making the Entity class itself more modular and flexible to the current game state.

- Singleton design pattern: The InteractionManager will be used as a script running throughout the entire function of the game in a single instance. It will be deferred to when needing to decide interactions as a way of mediating between the StatusEffect nodes and what should happen when one is added, and can be called on to do this process from anywhere.

API documentation:

- Entity Class - Basic class that holds generic information that could be used by either player or enemy, like speed, health, damage, idle movement, etc.
    - Parameters:
        - Max Health
        - Max Speed
        - Acceleration
        - Damage
        - Flat Defense
        - Percent Defense
        - Minimum Received Damage (This prevents immunity for both players and enemies if needed)
        - Ground Friction
        - Air Friction
    - Methods:
        - _Ready() - This will call the initial UpdateValues function to prepare for future behavior since by this point the child StatusEffect nodes, if there were any at start time, should have modified the multiplier and offset stats.
        - UpdateValues() - A function called whenever a change is made to one of the multipliers or offsets to calculate a new final value to be used at a later point.
        - Move() - This is a function that will use the Velocity parameter, apply its multipliers and offsets to that parameter, call the MoveAndSlide() function, and then set the Velocity property back to the original velocity. This separates the final offsets and multipliers from the initial velocity calculation that may exist within a player, so that the initial velocity calculations are not confused by the offset velocity from the last step, allowing for more accurate velocity tracking.
    - Signals:
        - Damage(double damageAmount, Vector2 knockback) - This signal will be emitted to the Entity object whenever it needs to receive damage from any source, whether it be an enemy, the player, or a StatusEffect node.
- StatusEffect Class - Holds information about what it will do, like doing some amount of damage after some amount of time, or adding a constant offset to player velocity, and then will modify its parent node’s attributes based on that information and apply damage through its own functions.
    - Parameters:
        - StatusEffectID (StringName)
        - Max Health Multiplier
        - Max Health Offset
        - Max Speed Multiplier
        - Max Speed Offset
        - Velocity Multiplier
        - Velocity Offset

        - Acceleration Multiplier
        - Acceleration Offset
        - Damage Multiplier
        - Damage Offset
        - Defense Multiplier
        - Defense Offset
        - Ground Friction Multiplier
        - Ground Friction Offset
        - Air Friction Multiplier
        - Air Friction Offset
        - Damage Over Time
        - Damage Interval (in Physics Ticks)
        - Duration/Lifetime
        - Transition (Boolean: If a secondary status will replace the old one if it has been applied for long enough)
        - Transition Effect
        - Transition Time (How long it takes for the transition to happen)
    - Methods:
        - _Ready() - Called on being added to the tree, this will update its parent's multipliers and offsets based on its own values, or it will clear itself from the tree if not attached to a member of the Entity class. This will then call the StatusEffectAdded function in the InteractionManager singleton.
        - _ExitTree() - Called once it is being removed from the tree and will undo any changes made to the parent Entity before calling UpdateValues on the parent Entity and being removed from the tree.
        - _PhysicsProcess(double delta) - Will call instantaneous or single instance functions that will modify the Entity state in the moment, and not over an interval. This will apply to functions like those that tick down the Entity parent's health. Will also tick down the remaining duration of the StatusEffect node to clear it naturally.
        - AttemptDamage(double delta) - Will be called in the physics process, tick down its current wait time until the next damage instance, and then when the remaining physics ticks until the next damage is done is zero or less, it will do a damage instance through the damage signal to the parent Entity and increase the current wait time by the Damage Interval's defined amount of ticks.
        - AttemptTransition(double delta) - Will be called in the physics process if transitions can occur, tick down its current wait time until the transition, and then when the remaining physics ticks until the transition is zero or less, it will transition to the defined transition effect.
- Recipe Resource:
    - Input Effects
        - Array of the effects required for this Recipe to be used.
    - Consumed Effects

        - This is an array of booleans that will be used to determine whether or not an input effect is removed upon this Recipe happening, or if it is kept.
    - Output Effects
        - Additional effects that are created separate from the input effects upon use of the Recipe.
    - Priority
        - If there are multiple interactions that can happen, do the Recipe with the highest priority, and if there is a tie in priority, do the Recipe that comes first in a deterministic fashion, i.e. alphabetical by name.
- InteractionManager Singleton:
    - Parameters:
        - File path to recipes folder
    - Methods:
        - LoadRecipes() - Used to initialize all of the Recipe resources into an array to be used throughout execution.
        - LoadStatusEffects() - Used to create a dictionary that associates every StringName of every StatusEffect node to its scene location within a status effect folder so that modifying a Recipe, or renaming a scene at some point does not require modifying every place that scene directory would be mentioned within Recipe resources or elsewhere.
        - StatusEffectAdded(Entity entity, StatusEffect newStatusEffect) - This will first check for if a StatusEffect node of the same type already exists under the parent Entity, and if it does, it will replace the duration of the old one with the longer of the two remaining durations. Then, this will run the newly added StatusEffect node against the Recipe list and check for existing combinations against the other statuses and then make the necessary combination, if it could react with multiple, react first based off of priority, then in the case of a conflict still, off of alphabetical by name. Then it will end with calling the UpdateValues function on the Entity.

Integration Approach with Godot:

- C++ module - This module will add additional nodes that can be added to scenes within the editor and can have their properties modified, or have their classes expanded with scripts just like any other node within Godot. It will also have resources that can be created in the editor that are later used for the Recipe system.
- Comparison with existing solutions (if applicable)
    - Not sure if there are any within Godot currently.

Game Design Document

Game Concept and Genre:

- WAKING OF THE WITCH is a side-scrolling platformer where the main objective of the game is to reach the end of the platformer without dying while fighting off enemies and avoiding obstacles.

  The game takes place in a world where magic is kept secret and separated from the mortal realm, but due to a new corrupt leadership taking control over the mortal realm, the government wants to utilize the magical realm for their own benefit. The player plays as a witch whose cat familiar got stolen by the government to power their weapons of mass destruction. To retrieve their magical partner-in-crime, the player will adventure from their home in the magical realm to the governmental base in the mortal realm where their cat familiar is being kept.

  On their journey, the player will stumble upon a variety of different areas that have their own environmental effects that the player has to take note of to bypass… The player starts off at their own abode in the magical realm. The magical realm is fairly peaceful, but hiding in the shadows lurks otherworldly entities, some dangerous others not. The realm that separates the magical and mortal realms is one of fire and incredible heat to ward off any prying mortal adventurers. To bypass, the player has to move strategically while avoiding fire and lava pits. When the player first enters the mortal realm, they exit from an ancient cave and are met with a hefty thunderstorming rainforest where they have to be careful of slipping in puddles and getting struck by lightning. To reach the government base, the player must make their way through many different biomes ranging from a dry, blazing desert to a smelly, bustling city. The government base is located in a vast tundra where the player has to be mindful of conserving their warmth and the strong wind that seems to be working with the government to prevent the player from reaching the base.

  The government has also managed to create their own weapons utilizing the magic they have stolen from the magical realm. This means their attacks/bullets do not only deal damage, but are also infused with magical powers that apply their own set of status effects on the player such as fire, poison, etc.. The player and enemies have ranged weapons (magic) to show off the status effects module and a fast-paced combat system.

Core gameplay mechanics:

- Movement: The player will move with WASD and Arrow Keys
  - Movement will be smooth, so that the fast-paced combat system works well and does not feel overwhelming/clunky to the player.
- Jump: The player will jump with the SPACE bar
  - The player will be able to jump fairly high to account for the magical capabilities of the character.
  - A Double-jump may be implemented to make the player feel more agile.

- Attack: Aim and Click with Mouse to attack and a projectile will be shot in the direction aimed.
    - A stronger attack may be implemented when the player holds down Mouse-click to charge their attack.
- Sprint: The player will sprint with the SHIFT key
- Dodge: A dodge system may be implemented to make the fast-paced combat flow better (double-tap direction of dodge, WASD or Arrow Keys)
- Core gameplay will mainly consist of traversing the game's world, changing strategy based on current effects due to enemy or environment.

How Does This Module Enhance Gameplay:

- Status effects make the playing environment more dynamic and reactive to what is going on in game. For example, if a player was hit with a fireball and simply took damage, it might feel alright to the player, but making the fireball actually set the player on fire can make the experience feel more realistic and immersive, improving the experience over simply taking damage. This could also apply to other effects made in other platformers, like a strong wind current pushing the player around which could be integrated as a status effect that modifies the player's speed by some offset.

Target Audience and Platform:

- The target audience is people who enjoy fast-paced action platformers.
- The game will be created with Godot, and will utilize a custom module created through Godot's GDExtension feature.

Visual Style and Aesthetic Direction:

- The game will be in a pixel art style similar to that of Celeste, so that it can emulate the magical feeling of the game world. We will use deep, saturated colors for a majority of the assets, but brighter colors of the main player and magic, to add contrast to the palette and to make the player easier to see. The backgrounds will hopefully be detailed, each with their own unique color palette, so that each environment is easily distinguishable and representative of the obstacles the player will encounter there.

Technical Implementation Plan

Development Timeline with Milestones:

- Week 1: Come up with solid Module Idea, start documentation
    - Initial ideation and concept creating
    - The broad overview of the module is understood
    - Begin creating general game ideas for showing off mechanics
- Week 2: Complete ½ of the documentation
    - The module specification should be mostly done
    - Concepting on the game should have significant work done
- Week 3: Documentation complete, start programming module
    - Module specification is done
    - Programming has started on the module
    - Game concept for demonstration is done
- Week 4: Work on module, start game development
    - Module has the core Entity and StatusEffect classes completed
    - Player, basic levels, and enemies have begun development
- Week 5: Finish Module, continue game development
    - Module is finished with fully implemented Entity and StatusEffect classes along with the Recipe resource and InteractionManager autoload script
    - Game has the basics of gameplay, art, and playable scenes
- Week 6: Finish game development, final cleanup
    - Game is in a complete state with a full, if even short, gameplay implementation
    - Focus is mostly on finishing the experience, polish, fixing issues, and completing art

Technology stack and dependencies:

- C#
- C++
- Godot standard libraries in C# and C++

Testing strategy and success metrics:

- For each type of system implemented, whether the interaction system, applying status effects, modifying player data, or otherwise, we will have scenes that individually test each feature, and will have a full test that sees that all systems are working together.
- For example, a unit test for applying status effects could work like this:
    - Start a scene and assert that the test Entity has properly loaded in.
    - Immediately after, add the StatusEffect node as a child node to the Entity and assert that it is a child of the Entity node.
    - After the StatusEffect node has had time to start up, compare the multipliers and offsets that exist in the Entity node against the modifications that the StatusEffect node was supposed to make, and assert that the default multiplier and offset amount plus the changes supposed to be made, are equal to the current multipliers and offsets.

- For this example, we are able to create a controlled environment and test if the system is working in isolation so that we can separate what might have gone wrong based on those tests. For example, if the expected change is correct after adding one StatusEffect, but wrong after adding two, there might be some problem with how the multipliers and offsets are being applied to the parent Entity that was unexpected.

Risk Assessment:

- In the case of something going wrong within the program, issues such as memory leaks, or total program crashes could happen if not managed properly.
- In the case of trying to handle extraordinarily large groups of entities, the amount of adding and removing children, physics update processes being handled, and interactions being cross checked between different Recipe resources could cause performance issues, causing the program to lag, and the user experience to be ruined.

Mitigation Strategies:

- Unit tests can be used to test features in isolation and ensure that each component in the system is behaving as expected, and additional tests can be created in order to test for combinations of behaviors happening in sequence, like multiple assignments of StatusEffect nodes, two components of a Recipe being added in different orders, or edge cases where effects might combine with another effect right as it is supposed to expire.
- Larger scale tests could also be made to benchmark changes, like simulating hundreds or thousands of entities all interacting with the status effect system at the same time and tracking performance improvements between versions. This can help to quantify changes as they happen, and give direction to optimizations being made.
- Profilers can be used in conjunction with large benchmarks to see what part of the system specifically, or what function, is taking up the most amount of computation time so that we have direction as to what might need to be looked at first for performance improvements instead of focusing on micro-optimizations.

Team Collaboration Plan

Role distribution and responsibilities

- Risa:
  - Github lead: Keeps Github organized
  - Programming: Game demo programming as needed, Entity class module component
- Jackson:
  - Documentation: Module and API documentation
  - Programming: Game demo programming as needed, InteractionManager and StatusEffect class module component
  - Art: Environmental art, backgrounds
- Maya:
  - Documentation: Game Design Documentation
  - Art: Character sprites (player and enemy), animations
  - Programming: Game demo programming as needed, Recipe resource module component

Communication Protocols and Meeting Schedule

- We will communicate about progress regarding the document, module progress, artistic progress, game progress, and other tasks, over our group Discord chat.
- We meet in person every Wednesday from 4-5pm every week leading up to the final submission date.

GitHub Workflow:

- Each member will work on their own branch, which is titled appropriately to what is being worked on that branch.
- Each branch should be an individual feature that is being made for the project, and will be merged into the project as it is completed and tested.
- We will require at least one other member to review the pull request and changes before merging.

Conflict Resolution Procedures:

- If there is a conflict when trying to merge, the merge requester will try to consolidate it. If they can’t find out what is wrong, help from other two members can be requested
- If there is conflict between two members, the third member mediates.

References and Resources

Academic and technical sources:

- Design Patterns

Tutorials and Documentation Consulted:

- Custom modules in C++ — Godot Engine (4.4) documentation in English

Inspiration and Prior Art

- Noita
- Celeste
- John Wick