6.3. CSC and CSU Descriptions

The main CSCI *Cosmonarium* is composed of multiple Computer Software Components, or CSCs. These include the Game Data Management CSC, which properly stores user save data; the Game Environment Management CSC, which is primarily responsible for displaying the game environment to the user as determined by user inputs and the supporting game logic; the Player Character CSC, which interprets user input and manipulates the in-game player characters; the Enemy Handling CSC, which manages enemy spawning behavior and movement within the overworld; the Story CSC, which controls scripted events and the dialogue that drives the story aspect of *Cosmonarium* forward; and the Music CSC, which controls which parts of the soundtrack are played at any given moment. Each of these CSCs is further broken down into Computer Software Units, or CSUs, and these CSUs will be further explored in the later sections of this document.

The Game Data Management CSC encompasses the Persistent Data Handling subsystem described in section 6.2.1.2 of the architectural design document. This CSC consists of a Save Manager CSU that saves and loads to persistent data, as well as a Persistent Data CSU that stores and returns the data for each node in the game that has information defined as persistent.

The Game Environment Management CSC controls which scene (and consequently, which enemies, objects, etc.) the player sees and can move around in at any time. This CSC primarily contains the Scene Manager CSU, which swaps between the numerous game scenes developed in the Godot Engine. The Scene Manager CSU is also in charge of switching the game mode from exploration to battle and vice versa based on in-game interactions.

The Player Character CSC runs everything related to the player-controlled characters in *Cosmonarium*. This includes the Input Engine CSU for interpreting user input and translating to on-screen movements; the Character Parent CSU, which defines the behavior and possible states that each of the three player characters may inherit; and the Party CSU, which manages all the collective data of the player characters.

The Enemy Handling CSC facilitates the spawning of enemies in the overworld and the transferral of their data to battle scenes. This CSC mainly consists of the Enemy Handler CSU, which defines and executes spawning behavior based on the current overworld scene; the Enemy Parent CSU, which defines the behavior and possible states that any standard enemies may inherit; the Boss Enemy CSU, which shares some similarities with the Enemy Parent CSU but instead stores boss enemy information in persistent data; and the Enemy Types CSU, which is a resource script containing all the possible enemy variations that can be spawned within *Cosmonarium*.

The Story CSC takes care of providing visual and textual information to the player in order to convey how the in-game story is progressing. This CSC includes the Sequencer CSU, which accepts a list of instructions and orders other CSUs across the project to execute these instructions; the Actor Engine CSU, which receives directions from the Sequencer and executes commands on the actors within the scene; the Camera Manager CSU, which pivots the player's viewport based on instructions from the Sequencer; the Event Sequences CSU, which is a collection of file paths to resource scripts that supply sequenced instructions to execute during cutscenes; and the Event Instructions Modules, which are these referenced files containing sequenced instructions.

The Music CSC directs how music is played based on changes within the game environment. This CSC primarily contains the Background Audio Engine CSU and the music files imported into the Godot Engine during *Cosmonarium's* development. The Background Audio Engine CSU switches music tracks in and out of Godot's audio playing system. Additionally, this CSU takes directions from both the Game Environment Management CSC and Story CSC to determine when to swap tracks.

6.3.1 Detailed Module Descriptions

The following sections provide details for all the main modules used in *Cosmonarium.*
Each module is effectively one of the CSUs mentioned above in section 6.3. Each of the following detailed module descriptions explains what purpose the module has and lists out all of that module's fields and methods with brief explanations. The module descriptions are roughly ordered from shortest to longest in terms of detail.

6.3.1.1 Event Sequences Module

The Event Sequences Module is a collection of file paths to resource scripts that supply instructions to be handed to and processed by the Sequencer and its associated partners. As such, the structure of this module is quite simple:

Event Sequences Fields:

- **sequences**: a dictionary of key-value pairs where each key is the name of a particular cutscene sequence and its associated value is the file path to the sequenced instructions for that cutscene.

Event Sequences Methods:

- **None**

6.3.1.2 Event Instructions Modules

The Event Instructions Modules are the resource scripts referred to by the Event Sequences Module. The modules in this category hold the scripted instructions needed by the Sequencer and its associated partners for executing in-game cutscenes. As such, the structure of these modules is quite simple:

Event Instructions Fields:

- **None**

Event Instructions Methods:

- **instructions()**: a static method returning an array of instructions to be parsed by the Sequencer and its associated partners in order to execute a cutscene within the game environment.

6.3.1.3 Main Scenes Module

The Main Scenes Module is a resource script containing references to all of the scene paths necessary in *Cosmonarium.* As such, the structure of this module is quite simple:

Main Scenes Module Fields:

- **explore_scenes:** a dictionary containing references to all the scenes needed in exploration mode. Each key is a colloquial name for the scene, and the value is the file path to that particular scene.
- **battle_scenes:** a dictionary containing references to all the scenes needed in battle mode. Each key is a colloquial name for the scene, and the value is the file path to that particular scene.

Main Scenes Module Methods:

- **None**

6.3.1.4 Enemy Types Module

The Enemy Types Module is a resource script providing references to files within the game that are necessary for spawning instances of enemies. As such, the structure of this module is quite simple:

Enemy Types Module Fields:

- **enemy_types:** a dictionary of dictionaries. Each key in the top-level dictionary is the name of an enemy type, and each value is a dictionary containing a String file path to the Godot scene for instancing that enemy, a String file path for instancing that same enemy's sprite for battle, and the base battle stats for that enemy type.

Enemy Types Module Methods:

- **None**

6.3.1.5 Entity Stats Module

The Entity Stats Module is a Godot class script defining an EntityStats object. An instance of this kind of object provides a comprehensive list of battle stats that can be used by a player character, enemy, or boss enemy. This following fields and methods can be found in this module:

Entity Stats Module Fields:

- **stat_keys:** an array of Strings, where each String value corresponds to a battle stat, e.g. HP, ATTACK, DEFENSE, etc.
- **stats_template:** a dictionary providing a template for a stats object. Each key is a String value from *stat_keys,* and each corresponding value is the integer 1. This field can help provide base stats for characters, enemies, and boss enemies that can be modified later.
- **LEVEL:** an integer value corresponding to the character or enemy's current level.
- **HP:** an integer value corresponding to the character or enemy's current health points. If this value drops to 0, the associated character is incapacitated.
- **MAX_HP:** an integer value corresponding to the character or enemy's maximum health points.The HP field must always be less than or equal to this field.
- **SP:** an integer value corresponding to the character or enemy's current skill points. If this value drops to 0, the associated character cannot use any of its special skills.
- **MAX_SP:** an integer value corresponding to the character or enemy's maximum skill points.The SP field must always be less than or equal to this field.
- **ATTACK:** an integer value corresponding to the character or enemy's base attack in battle.This field helps calculate how much damage can be dealt.
- **DEFENSE:** an integer value corresponding to the character or enemy's base defense in battle.This field helps calculate how much damage can be mitigated.
- **WAVE_ATTACK:** an integer value corresponding to the character or enemy's base wave attack stat in battle.This field helps calculate how much damage can be dealt in a wave attack.
- **WAVE_DEFENSE:** an integer value corresponding to the character or enemy's base wave defense stat in battle.This field helps calculate how much damage can be mitigated when faced with an opponent's wave attack.
- **SPEED:** an integer value corresponding to the character or enemy's base speed stat in battle.
- **WILLPOWER:** an integer value corresponding to the character or enemy's base willpower stat in battle.
- **LUCK:** an integer value corresponding to the character or enemy's base luck stat in battle.

Entity Stats Module Methods:

- **_init():** this method sets a character's stats to the base stats supplied in *stats_template* unless otherwise specified.
- **_set_stats():** this method takes in a dictionary containing stats fields and sets the character's current battle stats to the new stats dictionary.

- **_validate_stats():** this method takes in a dictionary containing stats fields and ensures that all stat fields have been initialized to an acceptable value (e.g. non-negative and below the capped value for each given field).
- **set_stat():** this method takes in a String referring to a certain stat and an integer value to assign to that stat for the instanced character. This method then sets the value of the stat key to the supplied integer value if the integer does not violate any rules regarding stat values, e.g. a character's HP value must be less than or equal to its MAX HP value.
- **to_dict():** this method returns a dictionary for the currently instanced EntityStats object.

6.3.1.6 Save Manager Module

The Save Manager Module is a singleton that controls how game data is saved to persistent data within a given user's machine. The contents of this module are utilized when the player reaches a point within *Cosmonarium* where he or she can save his or her progress. The following fields and methods can be found in this module:

Save Manager Module Fields:

- **save_files:** a constant array of Strings, where each String refers to a possible save file in *Cosmonarium*. Currently, *Cosmonarium* has three save files for regular users and three additional files for use in development.
- **current_save_index:** an integer value corresponding to the index of the file in *save_files* to which the user is currently accessing for saving or loading. During a given play session, a user can save to any file, as well as load from any file that has had data previously saved to it.
- **encrypt:** a boolean value corresponding to whether a save file being saved to or loaded from is intended to have some form of encryption applied to it.

Save Manager Module Methods:

- **_ready():** this method is called when the Save Manager singleton enters into Godot's SceneTree for the first time. This method ensures that update_persistent_data() is called whenever the Scene Manager emits the signal "goto_called." Essentially, the Save Manager updates the information to be stored in persistent data whenever directed to by the Scene Manager, which is typically during certain scene transitions.
- **save_game():** this method takes in a file name and writes the data currently defined in the Persistent Data module to a new file corresponding to the supplied name. This method writes to either an encrypted or unencrypted file, as specified by the *encrypt* field. After writing the current persistent data and file path to the current scene to the specified save file, the method closes the file.
- **update_persistent_data():** this method gathers all nodes in Godot's working SceneTree associated with the group "Persistent" (all game characters and objects with persistent data), and calls the save() function that each persistent node script contains. In this way,

all nodes supply their persistent data to the Persistent Data module, which later returns that same data to be written to a file in the Save Manager’s save_game() method.

- **load_game():** this method acts in conjunction with save_game() to allow players to resume gameplay where they last left off after saving. This method takes in a file name and opens the file with that name (or returns a debugging error if the file does not exist). The saved persistent data is loaded from the open file into each node it is associated with (e.g. character levels, whether bosses have been defeated or not, etc.), and the file path to the scene that the player was last in prior to saving is recovered. The save file is then closed, and the Scene Manager changes to the scene previously specified by the recovered file path, allowing the player to resume gameplay from where he or she last saved.
- **has_save_file():** this method takes in a String specifying the name of a save file and returns a boolean value corresponding to whether a save file of that name exists.

6.3.1.7 Persistent Data Module

The Persistent Data Module is a singleton that defines the main data structure used for storing persistent data as well as methods for manipulating that data and working with the Save Manager. The following fields and methods can be found in this module:

Persistent Data Module Fields:

- **data:** a dictionary containing the persistent data for all persistent nodes in *Cosmonarium*. Each key refers to a persistent object, and the value for each key is that object’s data to be saved (typically in a dictionary format itself, making *data* a dictionary of dictionaries).

Persistent Data Module Methods:

- **_ready():** this method is called when the Persistent Data singleton enters into Godot’s SceneTree for the first time. This method ensures that update_entry() is called whenever the Save Manager emits the “node_extracted” signal, and that restore_data() is called whenever the Scene Manager emits the “scene_loaded” signal.
- **deferred_restore():** this method calls restore_data() when it is safe to do so during the Godot Engine’s built-in physics process.
- **restore_data():** this method gathers all nodes in Godot’s working SceneTree associated with the group “Persistent” and loads the correct persistent data back into each node by calling _load_pdata() as needed. Additionally, if any of the persistent nodes possess the method on_load(), this method uses duck typing to call the on_load() method for each given node.
- **_load_pdata():** this method takes in a String ID value and an Object actor, and if *data* contains the provided ID, all the persistent data for the ID is loaded back into the corresponding actor.
- **update_entry():** this method takes in a dictionary containing a specific node’s persistent data and adds this data to the overall collection of persistent data in the *data* field. The

supplied dictionary is stored as the value in *data* with the node's persistence ID acting as the key.

- **get_data():** a getter method that returns the *data* object.
- **print_data():** this method prints out all the information in *data* by key.

6.3.1.8 Background Audio Engine Module

The Background Audio Engine Module is a singleton that determines which music from the game's soundtrack is played at any given moment. This module provides an auditory sensual experience to help immerse the player in the game environment. The following fields and methods can be found within this module:

Background Audio Engine Module Fields:

- **_music_player**: an AudioStreamPlayer node as defined by the Godot Engine. The *stream* property of _music_player refers to a music file that has been previously loaded and can be played.
- **tween_fade_out**: a Tween node as defined by the Godot Engine. This particular Tween is used to smoothly interpolate the Background Audio Engine's volume from its maximum value to its minimum value.
- **tween_fade_in**: a Tween object as defined by the Godot Engine. This particular Tween is used to smoothly interpolate the Background Audio Engine's volume from its minimum value to its maximum value.
- **transition_type_out**: an integer value corresponding to the type of transition that tween_fade_out uses. Currently for *Cosmonarium*, this value is 0, which refers to a linear transition from max volume to min volume.
- **transition_type _in**: an integer value corresponding to the type of transition that tween_fade_in uses. Currently for *Cosmonarium*, this value is 0, which refers to a linear transition from min volume to max volume.
- **min_volume_value**: a float value corresponding to the minimum value in decibels that the Godot Engine uses when playing back music tracks. Note: Godot's range for interpreting decibels is roughly between -60 dB (minimum audible) and 0 dB (maximum audible on hardware), so this field's value is typically set at -60.
- **max_volume_value**: a float value corresponding to the minimum value in decibels that the Godot Engine uses when playing back music tracks. Note: Godot's range for interpreting decibels is roughly between -60 dB (minimum audible) and 0 dB (maximum audible on hardware), so this field's value is typically set at 0.
- **current_song**: a String value corresponding to the song that the Background Audio Engine should be playing. This String is a file path to the actual music file to be played.

Background Audio Engine Module Methods:

- **facilitate_track_changes()**: this method takes in a String file path for a new possible song and calls request_playback() (and potentially request_playback_paused() if a song is already playing) with that file path.
- **request_playback()**: this method takes in a String file path for a new song, a boolean (with a default value of false) determined whether a fade in is unnecessary, as well as a fade in duration in seconds (with a default time of 1.0). This method loads and plays the new song, and then calls fade_in() if the provided boolean allows it.
- **request_playback_paused()**: this method takes in a boolean (with a default value of false) determining whether a fade out is unnecessary for the playing song, as well as a fade out duration in seconds (with a default value of 1.0). If a fade out is determined to be necessary, this method calls fade_out().
- **fade_out()**: this method takes in a float value for a time in seconds for fading out, and then starts tween_fade_out with the supplied properties to interpolate the currently loaded song. The thread running the Background Audio Engine yields until the fade out has completed.
- **fade_in()**: this method takes in a float value for a time in seconds for fading in, and then starts tween_fade_in with the supplied properties to interpolate the currently loaded song. The thread running the Background Audio Engine yields until the fade in has completed.
- **swap_songs_abrupt()**: this method takes in a String file path for a new song, abruptly stops any currently playing music, loads the new song into _music_player, and then plays that new song.

6.3.1.9 Camera Manager Module

The Camera Manager Module is a singleton that controls where the in-game camera is pointing, effectively determining which area of the game environment that the player can see at any given moment. This module most frequently works with the Sequencer to position the camera either to a newly specified location or to refocus back on the main player character party. The following fields and methods can be found within this module:

Camera Manager Module Fields:

- **camera:** a Camera2D node as defined by the Godot Engine. This field supplies the actual camera manipulated by the Camera Manager.
- **tween:** a Tween object as defined by the Godot Engine. This particular Tween is used to smoothly interpolate the position of *camera* from one Vector2 location to another.
- **viewport:** a Viewport node as defined by Godot. Basically, the player is restricted to seeing the game environment through the viewport, thus ensuring that only a section of the overworld can be viewed at any given time.
- **viewport_size:** a Vector2 that supplies the dimensions of *viewport*.
- **static_pos:** a Vector2 that focuses on the central point of the viewport, which is where the camera should be positioned when in the Static state.

- **State:** an enum that specifies the current state of the camera. Currently there are three separate states for the camera: OnParty, Sequenced, and Static.
- **state:** an enum value from *State* that describes the camera's current state.
- **y_cutoff:** an integer value that corresponds to how many pixels may be cut off in the vertical component of the viewport. Based on the size of a user's game window, some pixels may have to be cut off to ensure that the in-game pixel resolution is consistent.
- **vp_scale:** an integer value that corresponds to how much the viewport needs to be scaled depending on how a user has resized the game window.

Camera Manager Module Methods:

- **_ready():** this method is called when the Camera Manager singleton enters into Godot's SceneTree for the first time. This method adds *camera* and *tween* as children nodes of the Camera Manager and ensures that both operate correctly using Godot's built-in physics process. Finally, screen_resize() is called, with the guarantee that screen_resize() will be called again at any point the user attempts to resize the game window.
- **screen_resize():** this method takes the window size that the player is viewing the game through and ensures that the game viewport is properly scaled to this window. Thus. regardless of a user's computer screen resolution, the viewport should snap to fit the game window so that the user can comfortably view the game environment. Depending on the game window size, black bars may be set in the margins of the viewport (top and bottom, left and right) to preserve the in-game pixel resolution while fitting to the window.
- **_physics_process():** this method is a built-in Godot method that is called in the engine's physics process. This method properly aligns the camera for every step in the physics process, based on the value of *state.*
- **move_to_position():** this method takes in a Vector2 destination and a float value corresponding to a time (in seconds), and subsequently uses *tween* to interpolate the camera from its current position to the specified destination for the duration of the supplied time.
- **move_to_party():** this method takes in a float value of time (in seconds) and interpolates the camera from its current position back to the position in which it is locked onto the party for the duration of the supplied time.
- **release_camera():** this method returns the camera back to its static position after completing any sequenced camera instructions.
- **grab_camera():** this method checks if the camera is currently in the Sequenced state, and returns a boolean value based on this check. If the camera is not currently in the Sequenced state, it is switched into the Sequenced state and grabbed for use in a sequenced event. If the camera is already in use by a sequenced instruction, it cannot be grabbed again.
- **state_to_party():** this method sets *state* to OnParty.
- **state_to_static():** this method sets *state* to Static.

- **state_to_sequenced():** this method sets *state* to Sequenced.

6.3.1.10 Scene Manager Module

The Scene Manager Module is a singleton responsible for switching game environment scenes in and out to convey how the game world is structured. For example, when a player character steps into a doorway inside of a house, the Scene Manager transitions from showing the inside of the house to displaying the environment outside of the house. The following fields and methods can be found in this module:

Scene Manager Module Fields:

- **MainScenes:** a preloaded file providing access to an instance of the Main Scenes Module.
- **fade_screen:** a preloaded file that supplies a screen used for fading in and out during scene transitions.
- **current_scene:** a Node object for the current scene being displayed for the game environment; null if there is no scene currently within the SceneTree (which should occur rarely, if ever).
- **saved_scene_path:** a String containing the file path to a scene needed at a later point in time. For example, the current exploration scene can be saved when the game switches over to battle mode so that scene may be reloaded after the player successfully finishes the battle.
- **loader:** a field for holding a ResourceLoader object (as defined by Godot). This field is used to load resource files to be used interactively as scenes.
- **wait_frames:** an integer value providing how many of Godot's process frames need to pass before the loading animation can be safely played.
- **time_max:** an integer value corresponding to how many milliseconds the Scene Manager should attempt to wait while loading a new scene.
- **fade:** a field for holding an instance of *fade_screen* when needed.
- **warp_dest:** a String holding the warp destination of the player party in a new scene, if provided.
- **ticks:** an integer value for how many milliseconds have passed in the Godot Engine's process steps since the game was started. This value is used when trying to load a new scene file in _process().

Scene Manager Module Methods:

- **_ready():** this method is called when the Scene Manager singleton enters into Godot's SceneTree for the first time. This method gets the root node of the working SceneTree, and then saves the current scene of the game environment to (very aptly) *current_scene.*
- **goto_scene():** this method takes in a String scene ID, a String warp destination ID, and a boolean for whether the current scene path should be saved, and uses these passed arguments to determine which scene should be swapped in to replace the current scene.

The current scene path will be saved if indicated, and then the scene ID will be found in either the explore scenes or battle scenes listed out in the Main Scenes module. This method then calls _deferred_goto_scene() with the found scene path from the scene ID lookup (and the warp destination ID if one was supplied).

- **goto_saved():** this method calls _deferred_goto_scene() with whatever scene path has been stored in *saved_scene_path*.
- **_deferred_goto_scene():** this method takes in a String file path for a scene and a String warp destination ID, and switches scenes when it is safe to do so during Godot's process steps. This method plays the animation for *fade*, fading the screen to black while the current scene (soon to become the previous scene) is freed up. The new scene is then loaded temporarily into *loader,* and the method ensures that there are no errors with the newly loaded scene path. (Note that at this point, the player still can only see the fade screen.)
- **start_new_scene():** this method takes in a loaded scene file and creates a new instance of it, which is stored in *current_scene.* This new scene is added as a child to the working SceneTree. Additionally, the player character party is repositioned in the new scene if a warp destination ID had been previously specified in *warp_dest*. After the party is repositioned as needed, the animation for *fade* is played in reverse, and the newly loaded scene is made visible to the player. The fade screen instance is freed up, and gameplay resumes.
- **_process():** this method is a built-in Godot method that is called in the engine's process. This method checks if a new scene file has been loaded into *loader*, and waits for the loading animation to be prepared if this condition has been fulfilled. Then, for the amount of milliseconds specified in *time_max*, this method checks the loader, and if the resource file has successfully finished loading, start_new_scene() is called with the scene file.

6.3.1.11 Party Module

The Party Module is a script that orders the instanced player characters and handles all of the data owned collectively by them, such as items in the party inventory and who in the part has been incapacitated. The following fields and methods can be found within this module:

Party Module Fields:

- **actor_id:** a String value that identifiers the party as a group**.** This field is primarily used in the Actor Engine when executing commands meant for the whole party.
- **C2_in_party:** a boolean value for whether or not the secondary player character is in the party.
- **C3_in_party:** a boolean value for whether or not the tertiary player character is in the party.
- **active_player:** the player character currently to be controlled by the user via inputs. This character leads the other two members of the party during exploration.

- **party:** an array of player character names, based on which characters are currently instanced as children of the party node.
- **incapacitated:** an array of player charcter names, based on which characters have been incapacitated by the results of a previous battle.
- **items:** an array of item names, corresponding to the in-game items that the party currently has in its collective inventory.
- **spacing:** a float value that provides the amount of space that should be between each character in the party when they are in a following formation.
- **persistence_id:** the String value containing the party's persistence ID, used when storing the party's information to persistent data.

Party Module Methods:

- **sort_characters():** this method defines a sorting behavior for the characters in the party. Characters are sorted based on their ID; currently, the primary player character has an ID of "C1," and the other two characters are referred to by "C2" and "C3."
- **sort_alive():** this method defines a sorting behavior for the characters currently in the party. Characters are sorted based on whether or not they are still alive in the current scene, or if they have been incapacitated due to a previous battle.
- **init_in_party():** this method is called when a new character needs to be added to the party. Taking in a condition, a loaded character scene, and a name for the character, this method adds a new instance of the character scene and adds it as a child of the party node if the supplied condition has been fulfilled.
- **on_load():** this method is called when the party node enters the working SceneTree for the first time. The existing player characters meant to be in the party are sorted, and one character is designated the active player. The active player in the party is the character that the user controls via inputs. The remaining player characters follow behind the active player character.
- **reposition():** this method takes in a Vector2 corresponding to a new position, and also a new direction. This method sets the global position of the party to this new position, and ensures that each character faces in the new direction that has been provided.
- **save():** this method returns a dictionary of all the fields in the module that ought to be stored in persistent data.
- **change_sequenced_follow_formation():** this method takes in a String pertaining to how the party should act as a group and changes each player character's *sequence_formation* field to that string. This method is most often called for cutscene events. For example, if a cutscene is designed to have the player characters move independently of one another, an event instruction could pass in "split" as an argument to this method to get the desired player character behavior.

6.3.1.12 Boss Enemy Module

The Boss Enemy Module is an inheritable script that defines how boss enemies act within the overworld. Essentially, this module is a pared down version of the Enemy Parent Module, as bosses do not need autonomous movement options but they do need to store their information in persistent data. The following fields and methods can be found in this module:

Boss Enemy Module Fields:

- **default_speed:** an integer value providing the default speed at which an instanced boss can move in the overworld.
- **alive:** a boolean value corresponding to whether the current boss is still alive, based on the player's progress in the game plot.
- **persistence_id:** a String value for accessing and storing the boss's select information in persistent data.
- **skins:** a dictionary of dictionaries containing all the potential (overworld) sprites that the instanced boss may display.
- **animations:** the current AnimatedSprite (as defined by the Godot Engine) that is used when animating the boss's movements in the overworld.
- **stats**: the instanced boss's stats when transferred over to combat.
- **battle_id:** a String referring to the battle scene to switch to when swapping over from exploration mode to combat.
- **dir_anims:** a dictionary of arrays containing information on how to animate the boss in the overworld. The keys are the directions that the boss may face - up, down, left, and right - and the values are the animations that may be played for each direction.
- **current_dir:** the current direction that the boss is facing in the overworld.
- **isMoving:** a boolean value corresponding to whether or not the boss is currently moving around within the overworld.
- **velocity:** a Vector2 corresponding to the boss's current movement speed and direction.

Boss Enemy Module Methods:

- **initiate_battle():** this method adds the boss to the Enemy Handler's *queued_battle_enemies* and then calls the Scene Manager's goto_scene() method to automatically switch over to battle mode. This method will most often be called in sequenced events to start plot-driven conflicts.
- **_physics_process():** this method is a built-in Godot method that is called in the engine's physics process. Essentially, this method permits the instanced boss enemy to be moved around the game environment by the actor engine.
- **move_up():** this method moves the instanced boss up in the game environment when called by the Actor Engine.
- **move_down():** this method moves the instanced boss down in the game environment when called by the Actor Engine.
- **move_right():** this method moves the instanced boss right in the game environment when called by the Actor Engine.

- **move_left():** this method moves the instanced boss left in the game environment when called by the Actor Engine.
- **save():** this method returns a dictionary of the boss's information that should be saved to persistent data. The returned dictionary includes the boss's *persistence_id, position, current_dir, stats,* and *alive* fields.

6.3.1.13 Input Engine Module

The Input Engine Module is a singleton that interprets user inputs provided via external interfaces (e.g. a keyboard or gaming controller) and converts these inputs into player character actions. The following fields and methods can be found in this module:

Input Engine Module Fields:

- **to_player_commands:** a dictionary of dictionaries that maps keyboard or button inputs to player character actions. In the top-level dictionary, there are three keys: "pressed," "just_pressed," and "just_released." Each of these keys has a dictionary for a value, and each dictionary connects the key or button input to the appropriate method call.
- **to_menu_commands:** a dictionary of dictionaries that maps keyboard or button inputs to menu commands. In the top-level dictionary, there are three keys: "pressed," "just_pressed," and "just_released." Each of these keys has a dictionary for a value, and each dictionary connects the key or button input to the appropriate method call.
- **to_dialogue_commands:** a dictionary of dictionaries that maps keyboard or button inputs to dialogue actions. In the top-level dictionary, there are three keys: "pressed," "just_pressed," and "just_released." Each of these keys has a dictionary for a value, and each dictionary connects the key or button input to the appropriate method call.
- **to_battle_commands:** a dictionary of dictionaries that maps keyboard or button inputs to battle menu commands. In the top-level dictionary, there are three keys: "pressed," "just_pressed," and "just_released." Each of these keys has a dictionary for a value, and each dictionary connects the key or button input to the appropriate method call.
- **valid_receivers:** a dictionary of dictionaries containing all of the possible components of the game that can accept input. Each key pertaining to an input receiver (e.g. the battle menu, dialogue, player character, etc.) is mapped to a dictionary specifying in what priority given input is handled for that receiver, which process (regular or physics) takes care of that translated input's execution, and the specific translator for the given input. These translators refer to the four command dictionaries referenced above in this section/
- **input_disabled:** a boolean value for whether the Input Engine is processing any inputs provided by the user via external interfaces. User input is disabled for scene transitions and sequenced cutscenes.
- **input_target:** a field for holding the highest priority input receiver in *curr_input_receivers*, if *curr_input_receivers* is non-empty.

- **prev_input_target:** a field for holding the last stored value from *input_target*, if defined. Keeping track of the previous input helps prevent errors when the *input_target* is switched.
- **group_name:** a String with the value “Input_Receiver” that is used to add certain nodes to the group of the same name using Godot’s .add_to_group() method.
- **curr_input_receivers:** an array of all the nodes in the SceneTree that belong to the “Input_Receiver” group. This array may be empty or nonempty depending on the current state of the game environment.
- **disabled:** an array of input receivers that have been disabled, meaning they are not currently accepting any inputs to be translated.

Input Engine Module Methods:

- **_ready():** this method is called when the Input Engine singleton enters into Godot’s SceneTree for the first time. This method makes the user’s mouse invisible when within the game window, and also connects this module to the methods for disabling and reenabling input within the Scene Manager. Lastly, update_and_sort_receivers() is called.
- **update_and_sort_receivers():** this method gets all of the nodes from the SceneTree in the group “Input_Receiver” and stores these nodes in *curr_input_receivers*. If there is more than one input receiver node within *curr_input_receivers*, these input receivers are sorted in order of their priority.
- **activate_receiver():** this method takes in a node and, if that node is a valid input receiver, it is readded to the “Input_Receiver” group. The method update_and_sort_receivers() is called after this.
- **deactivate_receiver():** this method takes in a node and removes that node from the “Input_Receiver” group. The method update_and_sort_receivers() is called after this.
- **disable_input():** this method sets *input_disabled* to true.
- **enable_input():** this method sets *input_disabled* to false.
- **disable_player_input():** this method appends the “Player” input receiver to *disabled* to prevent any player input from being accepted and interpreted.
- **enable_all():** this method clears *disabled*, reenabling all input receivers.
- **_process():** this method is a built-in Godot method that is called in the engine’s process. This method calls process_input() by passing in “_process” as an argument.
- **_physics_process():** this method is a built-in Godot method that is called in the engine’s physics process. This method calls process_input() by passing in “_physics_process” as an argument.
- **sort_input_receivers():** this method defines the sorting behavior used for Godot’s .sort_custom() method in this module. Input receivers are sorted by the priority in which they should be handled. For example, when a menu is open, user inputs should only affect the menu. After the menu is closed, the user’s inputs can once again affect how her or his player party moves around the overworld.

- **process_input():** this method takes in a String specifying a loop type, which is either "_process" or "_physics_process." If *input_disabled* is true, *input_target* is null, or the ID of *input_target* is in *disabled*, this method returns without doing anything. Otherwise, this method checks whether the input target has changed (and handles this case if it has), and translate_and_execute() is called with the translator associated with the targeted input receiver.
- **transate_and_execute():** this method accepts one of the four dictionaries of dictionaries specified towards the beginning of this module's section, and uses this supplied translator to interpret user input as a command to be executed within the game environment.

6.3.1.14 Sequencer Module

The Sequencer Module is a singleton in charge of handling the sequenced cutscene events specified by each Event Instructions Module. The following fields and methods can be found within this module:

Sequencer Module Fields:

- **in_control:** a boolean value specifying whether or not the Sequencer is currently in control of the actors and actions of the current scene.
- **Events**: a preloaded file providing access to an instance of the Event Sequences Module.
- **active_event:** an array of instruction arrays if there is currently an active sequenced event queued up; null otherwise.
- **current_instruction:** the current instruction array pulled from *active_event* if *active_event* is non-null; null otherwise. Each instruction array specifies which instruction type it is (along with any needed parameters to execute the instruction) so that the Sequencer can pass the instruction to the correct method.

Sequencer Module Methods:

- **assume_control():** this method sets *in_control* to true and disables all user input interpretation in the Input Engine.
- **end_control():** this method sets *in_control* to false and reenables all user input interpretation in the Input Engine.
- **_process():** this method is a built-in Godot method that is called in the engine's process. The main logic of the Sequencer exists in this method, as it determines whether or not the Sequencer has control over the current scene and grabs the designated event as provided by the execute_event() method. This method then parses each instruction in *active_event* and passes it along to its respective instruction method, all of which are described below in this section.
- **execute_event():** this method takes in a String event ID and loads the resource script associated with that ID in *Events.* The method then assigns the array of instruction arrays to an appropriate variable and subsequently assigns a dictionary containing the event ID, the instructions, and the current instruction to *active_event*.

- **actor_instruction():** this method takes in an array of actor instruction parameters and passes the array along to a particular method in the Actor Engine on a case-by-case basis. The String value in the first position of the passed array determines whether the instruction is handled by the set, call, async or sync methods found within the Actor Engine. This method prevents the Actor Engine from accepting any new instructions if there are any asynchronous commands that have not finished yet; it also prevents the Sequencer from moving on to the next instruction until any ongoing synchronous command has been completed.
- **bg_audio_instruction():** this method takes in an array of audio instruction parameters and calls the Background Audio Engine method specified in the instruction's first index with any of the remaining parameters in the instruction. The method prevents the Sequencer from moving on to the next instruction until the current one has successfully been executed.
- **dialogue_instruction():** this method accepts a String dialogue ID and calls the Dialogue Engine's _beginTransmit() method with the supplied ID value.
- **battle_instruction():** this method accepts a String scene ID and calls the Scene Manager's goto_scene() method to facilitate a sequenced transition from exploration mode to battle mode. The method prevents the Sequencer from moving on to the next instruction until the new battle scene has fully loaded, though this instruction is also intended to be the last in any given active event sequence.
- **scene_instruction():** this method takes in a String scene ID and a String warp ID, and calls the Scene Manager's goto_scene() method with these parameters. The method prevents the Sequencer from moving on to the next instruction until the new scene has fully loaded, though this instruction is also intended to be the last in any given active event sequence. .
- **camera_instruction():** this method takes in an array of camera instruction parameters and either calls the Camera Manager's move_to_party() or move_to_position() method based on the value in the instruction array's first index. In either case, the method prevents the Sequencer from moving on to the next instruction until the camera has finished its repositioning.
- **delay_instruction():** this method takes in a float value specifying a time in seconds, and causes the Sequencer to yield for that amount of time. This functionality is especially useful in cutscenes where pauses are needed for dramatic effect.
- **signal_instruction():** this method takes in a String value corresponding to an object's ID as well as the name of a Godot-defined signal, and forces the Sequencer to wait until the object is observed to have emitted the specified symbol. This method offers usability for cases similar to delay_instruction().

6.3.1.15 Actor Engine Module

The Actor Engine Module is a singleton that accepts sequenced actor-based instructions from the Sequencer and executes those commands on the specified actor(s). The module works similarly to how a director works in a rehearsal for a play or music, calling out commands for the actors (player characters, specified enemies, NPCS, environment objects, etc) to enact based on behaviors that they already know and possess. Hence the Actor Engine utilizes duck typing in many of its methods. The following fields and methods can be found within this module:

Actor Engine Module Fields:

- **actors_dict**: a dictionary holding references to each actor in the current scene that can be given commands. For the key-value pairs in the dictionary, each key is the ID of an actor and the value is the actual node of the actor, such as a KinematicBody2D or RigidBody2D (as defined by the Godot Engine).
- **actors_array**: an array holding all the IDs of the actors in the current scene.
- **asynchronous_actors_dict**: a dictionary for all of the actors currently executing an async command in the scene. For the key-value pairs, the key is actual Node2D for the actor (e.g. KinematicBody2D, RigidBody2D, etc.), and the value is an array consisting of a SceneTreeTimer counting down the time left for the provided command to be executed, and a string of the command to be executed.
- **asynchronous_delay_time**: a float value for the longest amount of time (in seconds) for any currently executing async commands. When multiple async commands have been issued by the Actor Engine, all actors execute their specified methods for however long they have been directed to do so. The next instruction is only loaded from the Sequencer and parsed after a timer has timed out; this specific timer runs for however many seconds *asynchronous_delay_time* contains. The default assigned value prior to any async commands being parsed is 0.0 seconds.
- **sync_command_timer**: a SceneTreeTimer that is started (and restarted as needed after completion) for any sync command called on the current actor.
- **actor**: a Node2D for the current actor being handled by any of the Actor Engine's methods.
- **actor_position**: a Vector2 for the global position of the current actor being handled.
- **command_string**: a String value for the current command being parsed and handled by the Actor Engine; should refer to a method possessed by the *actor* field.
- **untimed_command_begun**: a boolean value referring to whether an untimed sync or async command has been called on the current actor.
- **untimed_command_done**: a boolean value referring to whether an untimed sync or async commanded that has already been called has completed.
- **additional_params**: a dynamically typed field for holding onto extra parameters an instruction may pass in for a command to be executed, e.g. a new position toward which the given actor should move.

Actor Engine Module Methods:

- **_ready**(): this method is called when the Actor Engine singleton enters into Godot's SceneTree for the first time. At such a point, this method should connect update_actors() to be called whenever the Scene Manager emits the "scene_loaded" signal, and also call update_actors() for the first time.
- **update_actors**(): this method gathers all nodes currently in the SceneTree that are associated with the Godot grouping "Actor" into *actors_array.* These nodes are then stored in *actors_dict* as appropriate. This method also causes any current async actors to execute their given commands as determined by the remaining time left on their associated SceneTreeTimers. If an async actor's timer has run out, this method removes that actor from *asynchronous_actors_dict.*
- **_physics_process**(): this method is a built-in Godot method that is called in the engine's physics process. This method facilitates the execution of any currently valid sync commands, as well as any untimed commands. If neither of these conditions are valid and relevant during the current physics process step and *asynchronous_actors_dict* is non-empty, update_actors() is called.
- **set_command**(): this method takes in a String *ID* for a given actor, a String *property* of that actor, and a dynamically typed *new_value* argument, and sets the specified property of that actor to the supplied new value.
- **call_command**(): this method takes in an ID, a name of a function, and an array of parameters, and calls the function on the actor with the ID given the supplied parameters. This method is mainly used for essentially instantaneous method calls, such as changing the sprite texture of a character.
- **async_or_sync_command**(): this method accepts a full instruction array from the Sequencer and parses the array contents to determine whether the passed instruction is asynchronous or synchronous as well as timed or untimed. Based on this determination, the method ensures that the command specified in the instruction is executed in its proper way, e.g. async timed, sync timed, async untimed, or sync untimed.
- **indicate_untimed_done**(): this command switches *untimed_command_done* to true when signalled to do so by the actor executing an untimed command.
- **execute_command**(): this command takes in an actor Node, a command String, and an array of additional parameters (with a default null value). Using all these parameters, the method uses duck typing to determine whether the supplied actor possesses the given command, and then calls that command on that actor (with any additional parameters) when it is safe to do so.
- **start_sync_command_timer**(): this method takes in a float value (in seconds) and assigns a new SceneTreeTimer to *sync_command_timer* with the supplied number of seconds.
- **add_actor_to_asynchronous_actors**(): this method takes in an actor Node, a command String, and a float value for time, and adds the actor to *asynchronous_actors_dict* as a

key with the value of an array containing a SceneTreeTimer with the supplied number of seconds at one index and the command string at the second array index.

6.3.1.16 Enemy Handler Module

The Enemy Handler Module is a singleton that controls how enemy characters spawn within each exploration scene. The following fields and methods can be found within this module:
Enemy Handler Module Fields:

- **Enemies:** a preloaded resource file supplying all possible enemies for spawning; see the Enemy Types Module for more detailed information.
- **spawn_body:** an invisible test sprite used to check where enemies can be spawned in the current game environment.
- **random_num_generator:** a random number generator to aid in calculating when and where to spawn enemies.
- **generated_enemy_id:** the current integer ID value of a generated enemy; the starting value is always 1.
- **num_enemies:** the integer value for the number of enemies currently spawned within the scene. This information is gathered from the Explore Root of the current scene upon its load.
- **max_enemies:** the integer value for the maximum number of spawnable enemies within the current scene. This information is gathered from the Explore Root of the current scene upon its load.
- **target_player:** the KinematicBody2D of the character in the player's party that enemies will target and chase.
- **player_view:** a Vector2 for the size of the viewport that the player can see the game environment, as provided by the Camera Manager.
- **tilemap_rect:** a Rect2 object (as defined by the Godot Engine) corresponding to the tiles used within the current scene's tilemap.
- **tilemap_reference_point:** a Vector2 corresponding to the central position of the current scene, as translated from tilemap coordinates to global coordinates.
- **tilemap_dimensions:** a Vector2 corresponding to the width and height of the current scene's tilemap.
- **tilemap_width:** an integer value for the width of the given tilemap, as provided by *tilemap_dimensions*.
- **tilemap_height:** an integer value for the height of the given tilemap, as provided by *tilemap_dimensions.*
- **spawn_area:** an Area2D that encompasses all locations in the current scene where enemies may be spawned.
- **spawn_collider:** a CollisionPolygon2D object that covers the entire polygonal area of *spawn_area*
- **scene_node:** a reference to Node of the current scene, as provided by the Scene Manager.

- **existing_enemy_data:** a dictionary of currently existing enemies, where the key is a given ID as determined by *generated_enemy_id* and the value is the corresponding data from *Enemies* for the given spawned enemy type.
- **queued_battle_enemies:** an array of enemies queued up in exploration mode to be transferred over to battle mode for instancing.
- **queued_despawns:** an array of enemies queued up in battle mode to be transferred over to exploration mode for despawning.
- **can_spawn:** a boolean value corresponding to whether enemies can be spawned in the current scene or not. This value is set from the Explore Root of the current scene upon its load.
- **spawning_locked:** a boolean value corresponding to whether the spawn_enemy() function should be called or not. Only one enemy should be spawned at a time; thus, this field is reminiscent of a mutex.
- **viewport_buffer:** an integer buffer that adds to the "size" of the player's viewport. Enemies should only be spawned in locations outside of the player's viewport plus the value of this field.
- **valid_pos_flag:** a boolean value corresponding to whether an enemy can be spawned in a specified position within the game environment.

Enemy Handler Module Methods:

- **_ready():** this method is called when the Enemy Handler singleton enters into Godot's SceneTree for the first time. This method checks if enemies can be spawned in the current scene and calls initialize_spawner_info() if spawning is possible.
- **get_enemy_data():** this method takes in an integer ID value and returns the data in *existing_enemy_data* corresponding to that ID.
- **intialize_spawner_info():** this method gathers dimensional and positioning data from the tilemap of the current scene in order to determine where enemies may be spawned.
- **spawn_enemy():** this method attempts to spawn an enemy at a randomized position within the current scene. After getting such a position from calling get_spawn_position(), this method calls validate_spawn_position() and checks if the supplied position lies within the player's current viewport. If both checks are passed, the method spawns a new enemy at the given position after updating *generated_enemy_id* and *existing_enemy_data.* This new enemy node is added as a child to the current scene in Godot's working SceneTree.
- **get_spawn_position():** this method returns a randomized position (a Vector2) within the confines of the tilemap, <u>not</u> just the spawn area.
- **validate_enemy_spawn():** this method takes in a possible spawning location, and, using *spawn_body* as a test, checks whether the location lies within the *spawn_collider* but outside of the player's viewport.

- **set_valid_pos_flag():** this method is used within validate_enemy_spawn() to see if *spawn_body* enters into *spawn_area*. If this entry is detected, *valid_pos_flag* is set to true.
- **reset_valid_pos_flag():** this method is used within validate_enemy_spawn() if *spawn_body* ever exits *spawn_area*. If this exit is detected, *valid_pos_flag* is set to false.
- **despawn_enemy():** this method despawns enemies specified in *queued_despawns* following a player's victory in battle mode. Each enemy's despawning animation is played, and then it is removed from the working SceneTree in the Godot Engine.
- **_physics_process():** this method is a built-in Godot method that is called in the engine's physics process. For the Enemy Handler, this method checks if enemies can be spawned in the current scene, and then calls spawn_enemy() until *num_enemies* equals *max_enemies* or the Enemy Handler is otherwise told to stop spawning new enemies.

6.3.1.17 Enemy Parent Module

The Enemy Parent Module is an inheritable script that defines how enemies act within the overworld. This module is inherited by and associated with instances of enemies spawned by the Enemy Handler. The following fields and methods can be found in this module:

- **Enemy Parent Module Fields:**
    - **Mode:** an enum that specifies which behavioral pattern an instance enemy should follow. Currently there are four separate modes for an enemy: Stationary, Chase, Patrol, and Battle.
    - **default_speed:** an integer value providing the default speed at which an instanced enemy can move in the overworld.
    - **chase_factor:** a float value for optionally modifying an instanced enemy's chasing speed.
    - **alive:** a boolean value corresponding to whether the current instanced enemy is still alive.
    - **current_mode:** an enum value from *Mode* that describes the instanced enemy's current behavioral pattern.
    - **data_id:** an integer value assigned to the instanced enemy by the Enemy Handler upon spawning.
    - **has_patrol_pattern:** a String describing the type of patrolling pattern an instanced enemy follows.
    - **player_party:** a field for storing the existence of the player character party whenever a player character has entered an instanced enemy's area of detection.
    - **target_player:** the KinematicBody2D of the character in the player's party (as provided by the Enemy Handler) that an instanced enemy will target and chase.
    - **skins:** a dictionary of dictionaries containing all the potential (overworld) sprites that the instanced enemy may display.
    - **animations:** the current AnimatedSprite (as defined by the Godot Engine) that is used when animating the instanced enemy's movements in the overworld.

- **patrol_patterns:** an array of Strings defining types of patrol behavior that enemies may follow. Currently, there are four patrol types to be implemented in the future: erratic, linear, circular, and rectangular.
- **dir_anims:** a dictionary of arrays containing information on how to animate an enemy in the overworld. The keys are the directions that an enemy may face - up, down, left, and right - and the values are the animations that may be played for each direction.
- **current_dir:** the current direction that the enemy is facing in the overworld.
- **isMoving:** a boolean value corresponding to whether or not the instanced enemy is currently moving around within the overworld.
- **rand_num_generator:** a random number generator for randomizing certain times and values for the instanced enemy.
- **initial_mode:** the *Mode* value that an instanced enemy starts out with after spawning.
- **velocity:** a Vector2 corresponding to the enemy's current movement speed and direction.
- **patrol_timer:** a SceneTreeTimer that runs with a randomized time. This timer determines how long an instanced enemy should execute a patrol movement method.
- **wait_timer:** a SceneTreeTimer that runs with a randomized time. This timer determines how long an instanced enemy should wait before choosing its next movement in the erratic patrol method.
- **next_movement:** a String that describes which movement method should next be executed by an instanced enemy.

- **Enemy Parent Module Methods:**
  - **_ready()**: this method is called when an instanced enemy is added to Godot's working SceneTree. Specifically, this method connects the enemy's chasing methods to signals emitted by the associated Area2D nodes used for detecting whether a player character is close enough to chase.
  - **_physics_process():** this method is a built-in Godot method that is called in the engine's physics process. This method directs an instanced enemy how to move based on whether it is currently in Stationary, Chase, or Patrol mode. Additionally, the logic for generally transitioning from exploration mode to battle mode exists in this method: game modes shall be switched from the former to the latter when an instanced enemy collides with a member of the player character party.
  - **move_toward_player():** this method modifies the instanced enemy's movement vector so that it points towards the target player's current position in the game environment.
  - **move_up():** this method moves the instanced enemy up in the game environment when called by the enemy itself or by the Actor Engine.
  - **move_down():** this method moves the instanced enemy down in the game environment when called by the enemy itself or by the Actor Engine.
  - **move_right():** this method moves the instanced enemy right in the game environment when called by the enemy itself or by the Actor Engine.

- **move_left():** this method moves the instanced enemy left in the game environment when called by the enemy itself or by the Actor Engine.
- **begin_chasing():** this method switches the instanced enemy's *current_mode* field to "Chase" when a player character enters into its detection field.
- **stop_chasing():** this method switches the instanced enemy's *current_mode* field from "Chase" to the value of *initial_mode* when a player character exits its detection field.
- **patrol_erratic():** this method (in tandem with _physics_process()) allows instanced enemies to randomly walk around the game environment.
- **patrol_linear():** this method allows instanced enemies to patrol between two specified points within the game environment.
- **patrol_box():** this method (in tandem with _physics_process()) allows instanced enemies to patrol in a specified rectangular path within the game environment.
- **patrol_circle():** this method (in tandem with _physics_process()) allows instanced enemies to patrol in a specified circular path within the game environment.

6.3.1.18 Character Parent Module

The Character Parent Module is an inheritable script that defines how player characters may act within the overworld. Due to the versatility needed for player characters in the overworld, this module is one of the largest in the overall project. The following fields and methods can be found in this module:

Character Parent Module Fields:

- **pixels_per_frame:** the number of pixels a player character can move per frame.
- **default_speed:** the default speed at which a player character may move.
- **speed:** initially *default_speed*, an instanced player character's speed may be modified during cutscenes.
- **persistence_id:** a String value that supplies the ID used when storing the instanced character's information into persistent data.
- **input_id:** a String value that supplies the ID used when executing inputs translated by the Input Engine for the active player.
- **actor_id:** a String value that supplies the ID used when the Actor Engine is executing sequenced commands on the instanced player character.
- **alive:** a boolean value for whether or not the instanced player character is alive in the current scene.
- **exploring:** a boolean value for whether the instanced player character is in exploration mode or not within the current scene.
- **skills:** a dictionary containing all of the skills that the instanced character possesses.
- **stats:** an instance of the EntityStats class as defined by the Entity Stats module.
- **skins:** a dictionary of dictionaries containing all the potential (overworld) sprites that the instanced player character may display.

- **current_skin:** a String value corresponding to which graphical skin the character should be displaying. This field is used when accessing *skins* to set the value for *animations*.
- **animations:** the current AnimatedSprite (as defined by the Godot Engine) that is used when animating the instanced player character's movements in the overworld.
- **interact_areas:** an array of game objects within the environment with which an active player character may interact.
- **party_data:** a dictionary containing the data passed down from the parent party node to the instanced character child node.
- **velocity:** a Vector2 corresponding to the character's current movement speed and direction.
- **isMoving:** a boolean value corresponding to whether or not the instanced player character is currently moving around within the overworld.
- **current_dir:** the current direction that the character is facing in the overworld.
- **dir_anims:** a dictionary of arrays containing information on how to animate a player character in the overworld. The keys are the directions that a character may face - up, down, left, and right - and the values are the animations that may be played for each direction.

Character Parent Module Methods:

- **on_load():** this method is called when the instanced character node enters the working SceneTree for the first time. This method sets the player character's position and ensures that the character's current skin is the default.
- **_physics_process():** this method is a built-in Godot method that is called in the engine's physics process. If *exploring* is set to true, this method calls the explore() method for the instanced character.
- **play_anim():** this method takes in a String specifying a specific animation and plays that animation.
- **set_anim():** similar to the above method, this method takes in a String specifying a specific animation and sets the *animation* property or *animations* to the specified string.
- **flip_horizontal():** this method takes in a boolean that determines whether or not a flip should occur. In this way, only the left or right animations need to be defined for the player character, and the reverse direction can just be flipped.
- **explore():** this method handles how instanced player characters act during exploration mode. Inactive characters are added as party members following the active player. Meanwhile, the active player accepts inputs from the user to determine how it should move within the game environment. This method handles that movement as well as the animations that play alongside it.
- **activate_player():** this method sets the instanced player character as the receiver for user inputs and also enables the character's collision box. This method is used when switching a new player character to the active player.

- **deactivate_player():** this method removes the instanced player character from being the receiver for user inputs and also disables the character’s collision box. This method is used when switching a new player character from being the active player to being a following party member.
- **set_collision():** this method takes in a boolean for whether or not a character’s collision box is enabled and negates that boolean, assigning it to the collision box’s *disabled* property for that character.
- **move_up():** this method moves the instanced player character up in the game environment when called by the character itself or by the Actor Engine.
- **move_down():** this method moves the instanced player character down in the game environment when called by the character itself or by the Actor Engine
- **move_right():** this method moves the instanced player character right in the game environment when called by the character itself or by the Actor Engine
- **move_left():** this method moves the instanced player character left in the game environment when called by the character itself or by the Actor Engine
- **open_menu():** this method calls the activate() method in the Menu Manager, opening up the main menu.
- **interact():** this method allows the active player to interact with
- **change_follow():** this method takes in a String pertaining to how the character should act within the party and changes the instanced character’s *sequence_formation* field to that string value. This method is most often called for cutscene events when player characters need to move independently from one another.
- **set_speed():** this method takes in a float value and sets the character’s *speed* field to this new float value.
- **restore_speed():** this method resets the character’s *speed* field to *default_speed.*
- **scale_anim_speed():** this method takes in a float value for scaling the character’s animation speed, and sets the speed scale to the provided float.
- **restore_anime_speed():** this method resets the character’s animation speed to 1.0, which means that the character’s animations play at the regular speed.
- **change_skin():** this method takes in a String specifying an instanced character’s skin and switches the displayed skin to the one supplied.
- **move_to_position():** this method takes in a new position vector and a boolean referring to whether the position passed in is a global position or not. This method then moves the player character to the new position by playing its walking animations from its starting position to the ending position. This method is primarily used in sequenced events when player characters need to be moved to a new location within an unspecified amount of time.
- **save():** this method returns a dictionary of all the fields in the instanced player character node that ought to be stored in persistent data.

- **follow():** this method handles the following behavior of inactive party members. This method properly spaces the following player characters apart from one another and the party leader.

6.3.2 Detailed Interface Descriptions

Given the numerous moving parts involved in the project, each CSU communicates and shares information with at least one other CSU. In this case, modules transfer data with other modules typically by accessing each other's fields or calling each other's methods. The various ways in which each module interfaces with the other modules relevant to it are described in the following subsections.

6.3.2.1 Event Sequences

6.3.2.1.1 Event Sequences ← Event Instructions

The Event Instructions Modules all directly provided file paths that the Event Sequences module can pass over to the Sequencer. Thus, the Event Sequences module is simply a middleman between the Sequencer and each Event Instructions module, offering a simplified way of accessing them all.

6.3.2.2 Event Instructions

The Event Instructions modules do not rely on any other modules for the services they provide, as each is simply a resource file containing an array of sequenced event instruction arrays.

6.3.2.3 Main Scenes

As a resource file, the Main Scenes module supplies information to the Scene Manager, and does not depend on any other modules itself. This module, as the name of its file type suggests, is a resource and not a script describing any real functionality or behavior necessary within the game environment.

6.3.2.4 Enemy Types

6.3.2.4.1 Enemy Types ← Entity Stats

When the Enemy Types module defines the data for each type of enemy, it requires Entity Stats to supply base stats. Each enemy type has a specific set of base stats. These base stats may be modified slightly when instancing each type of enemy to add some variation to battles.

6.3.2.5 Entity Stats

As a class definition file, the Entity Stats module does not depend on any other modules to do its work. Instead, it acts as a support for instancing player characters, enemies and boss enemies within the game by supplying consistent data structures containing battle stats.

6.3.2.6 Save Manager

6.3.2.6.1 Save Manager ← Scene Manager

Whenever the Scene Manager emits the “goto_called” signal, the Save Manager calls its update_persistent_data() method. This connection between the two modules ensures that persistent data is up-to-date every time a new scene in the game environment is loaded up. Additionally, whenever the Save Manager loads a particular save file, it calls on the Scene Manager to go to the scene specified by the save scene file path.

6.3.2.6.2 Save Manager ← Persistent Data

When saving game data to a file, the Save Manager accesses the *data* dictionary in the Persistent Data module. The Save Manager then writes the persistent data it has grabbed to the proper save file and closes that file after completing its write operation.

6.3.2.7 Persistent Data

6.3.2.7.1 Persistent Data ← Scene Manager

Whenever the Scene Manager emits the “scene_loaded” signal, Persistent Data calls its restore_data() method. This connection between the two modules ensures that the correct persistent data is loaded into all the nodes in a scene following a scene change.

6.3.2.7.2 Persistent Data ← Save Manager

Whenever the Save Manager emits the “node_data_extracted” signal, Persistent Data calls its update_entry() method. The Save Manager emits this specific signal within its update_persisten_data() method, and passes along the node data that it has extracted from each specified persistent node. Connecting this signal to update_entry() in Persistent Data seamlessly stores the newly extracted data into *data*, the overall dictionary of persistent data.

6.3.2.8 Background Audio Engine

6.3.2.8.1 Background Audio Engine ← Scene Manager

Although the Background Audio Engine does not technically rely on a direct connection to the Scene Manager, it does depend on the current scene that has been loaded up. The Background Audio Engine grabs the information regarding the track it is meant to play primarily from the general explore root associated with each scene.

6.3.2.9 Camera Manager

6.3.2.9.1 Camera Manager ← Scene Manager

The Camera Manager communicates with the Scene Manager to determine what the bounds are for the camera in the current scene. In this way, the current scene dictates how much of the game environment the player can see at any given time, which adds flexibility to the game design.

6.3.2.10 Scene Manager

6.3.2.10.1 Scene Manager ← Main Scenes

The Scene Manager has few other modules that it depends on for it to function properly, and instead many other modules are dependent on knowing what the current scene is. However, the Scene Manager does rely on a preloaded instance of the Main Scenes module in order to easily check and see which scenes are within the scope of *Cosmonarium*.

6.3.2.11 Party

The Party module indirectly works with Scene Manager, as the members of the party are somewhat determined by what the current scene is. However, this module has no direct connection with any of the other modules specified in this document. Instead, the Party module works closely with the SceneTree to add and remove instanced player characters as child nodes.

6.3.2.12 Boss Enemy

6.3.2.12.1 Boss Enemy ← Enemy Handler

Each boss enemy, when confronting the player in a cutscene, must somehow transfer its data over to the boss battle scene. This transfer occurs using the *queued_battle_enemies* array found in the Enemy Handler module. Each boss's persistent ID is appended to this array prior to its respective boss battle, and the battle sequence extracts the boss's battle data from persistent data using that transferred persistence ID.

6.3.2.12.2 Boss Enemy ← Scene Manager

After appending its persistence ID to *queued_battle_enemies*, a boss calls on the Scene Manager to transition from the current exploration to the boss battle scene. The Boss Enemy therefore depends heavily on the Scene Manager for smooth, dramatic transitions from sequenced events to actual combat mode.

6.3.2.12.3 Boss Enemy ← Persistent Data

Each boss saves its information to the *data* dictionary found in the Persistent Data module, as bosses are unique and should only be fought once within each playthrough of the game.

6.3.2.13 Input Engine

6.3.2.13.1 Input Engine ← Scene Manager

The Scene Manager plays a crucial role in how the Input Engine works. When the Scene Manger begins transitioning to a new scene, the Input Engine is signalled to call disable_input(), thereby ignoring any user input during the transitionary period. Once the new scene has partially loaded, the Scene Manager directs the Input Engine (again by way of a Godot signal) to call update_and_sort_receivers(). This command is issued so that only the input receivers present in the new scene can be affected by user input; all old and irrelevant input receivers are thrown out. After the new scene fully loads, enable_input() is called by the Input Engine, and user input once again is transformed into character actions on-screen.

6.3.2.14 Sequencer

6.3.2.14.1 Sequencer ← Event Sequences

The Sequencer preloads an instance of the Event Sequences module in order to have access to all of the possible events that may occur in *Cosmonarium*. When given the name of an event, the Sequencer looks up and the event in this preloaded file and gets back an array of sequenced instruction arrays.

6.3.2.14.2 Sequencer ← Input Engine

The Sequencer is able to execute its assume_control() and end_control() methods by calling on the Input Engine to execute its own disable_player_input() and enable_all() methods, respectively. When the Sequencer intends to parse through and execute sequenced instructions, user input must be blocked to avoid messing up the events. Conversely, when a cutscene has finished, control over the active player character must be returned to the user, and all other input receivers must also be reenabled.

6.3.2.14.3 Sequencer ← Actor Engine

Whenever the Sequencer comes across instructions for a specific actor in the scene, it passes the instruction array to the Actor Engine. The Actor Engine then handles the supplied instructions as it sees fit.

6.3.2.14.4 Sequencer ← Background Audio Engine

Whenever the Sequencer comes across instructions for changing the audio track playing in the background, it passes the instruction array to the Background Audio Engine. The Background Audio Engine then strips out the values within the instruction array that it needs, and handles them as it sees fit.

6.3.2.14.5 Sequencer ← Dialogue Engine

Whenever the Sequencer comes across instructions for displaying some dialogue text, it calls the Dialogue Engine with the argument supplied with the instruction array. The Dialogue Engine takes in the dialogue ID argument and returns by displaying the selected text to the user.

6.3.2.14.6 Sequencer ← Scene Manager

Whenever the Sequencer comes across instructions for switching to a new scene, it passes the scene ID (and warp ID if applicable) to the Scene Manager. The Scene Manager then uses these IDs to change over to the newly desired scene.

6.3.2.14.6 Sequencer ← Camera Manager

Whenever the Sequencer comes across instructions for modifying the camera position, it passes the instruction array to the Camera Manager, The Camera Manager then interprets the arguments passed in the array to call its own methods.

6.3.2.15 Actor Engine

6.3.2.15.1 Actor Engine ← Scene Manager

The Actor Engine most directly relies on the Scene Manager module to maintain an accurate list of actors within the scene. Whenever a new scene is loaded, the Actor Engine is signalled to call update_actors() to make sure that the most update-to-date list of actors can be stored in *actors_array.*

6.3.2.16 Enemy Handler

6.3.2.16.1 Enemy Handler ← Enemy Types

The Enemy Handler relies on the Enemy Types module to provide a list of enemy types that can be spawned into any given scene. To facilitate this dependency, the Enemy Handler preloads an instance of the Enemy Types resource script and assigns it to the *Enemies* field, where it can be referenced as needed.

6.3.2.16.2 Enemy Handler ← Scene Manager

The Scene Manager provides the Enemy Handler with the name of the current scene. Using this name, the Enemy Handler can access the explore root of the current scene, which holds information about whether or not enemies can be spawned, and the maximum number of spawnable enemies if spawning is permitted. Also, after a battle has been completed, the Enemy Handler must wait for the Scene Manager to reload the previous exploration scene before despawning defeated enemies.

6.3.2.16.3 Enemy Handler ← Camera Manager

The Enemy Handler relies on the Camera Manager to get access to the player's viewport. Enemies can only be spawned outside of the player's view (for the sake of fairness and aesthetics), and so the Camera Manager provides the Enemy Handler with the viewport size. From this information, the Enemy Handler can calculate whether an enemy spawn position lies inside or outside of the player's view at any given time.

6.3.2.16.4 Enemy Handler ← Party

The Party module gives the Enemy Handler information about which player character is currently active, and the Enemy Handler stores this information in its *target_player* field.

6.3.2.17 Enemy Parent

6.3.2.17.1 Enemy Parent ← Enemy Handler

The Enemy Handler module passes on the data in its *target_player* field to each instanced enemy. In this way, each instanced enemy knows which player character it should target and chase within the overworld. Additionally, whenever an instanced enemy makes contact with a player character, it calls the Enemy Handler to prevent any further enemies from spawning. The enemy that has collided with the player also has the Enemy Handler add it to the *queued_battle_enemies* array so that its data can be transferred over to the battle scene.

6.3.2.17.2 Enemy Parent ← Scene Manager

After having its data added to *queued_battle_enemies,* the instanced enemy that has collided with the player character calls on the Scene Manager to transition from the current exploration scene to a generic battle scene.

6.3.2.18 Character Parent

6.3.2.18.1 Character Parent ← Entity Stats

The Character Parent module utilizes the Entity Stats class to provide each instanced player character with stats for battle. Each instanced player character starts out with the base stats given in the *stats_template* dictionary in Entity Stats. Over the course of the game, however, each character will improve the values of their stats by winning tougher and tougher battles.

6.3.2.18.2 Character Parent ← Party

The Character Parent receives the information in its *party_data* field from the Party module's on_load() method. The Party module gives every player character instance a dictionary to store in *party_data* and this dictionary includes data about who the active player character is, what position the current character is in the party, an array of all the characters in the party, how far apart the current character is spaced from the other instanced party members, and whether or not the current character ought to be following or moving independently (in the case of a sequenced event).

6.3.2.18.3 Character Parent ← Input Engine

An instanced player character relies on the Input Engine to call its activate_receiver() and deactivate_receiver() methods in order for the player character to call its own activate_player() and deactivate_player() methods, respectively. The Input Engine directs all translated user input to whichever player character is currently the active one, and thus this connection between the two modules is intuitive.

6.3.2.18.4 Character Parent ← Menu Manager

The active instanced player character uses the functionality found within the Menu Manager to access the main menu and its various submenus. With its open_menu() method, the active instanced player character directs the Menu Manager to call its own activate() method, thereby allowing the player to view the party's inventory, player statuses, etc.

6.3.2.18.5 Character Parent ← Persistent Data

Each instanced player character stores its data (skills, status of being alive, level and current stats, etc.) in the *data* field found in the Persistent Data module.

6.3.3 Detailed Data Structure Descriptions

A majority of the data structures utilized in the above CSU modules are standard arrays and dictionaries, and have already been mentioned in each CSU's list of fields. However, the information regarding these data structures is repeated in the following section for extra context and ease of lookup.

6.3.3.1 Event Sequences

The Event Sequences module acts as a go-between for the Sequencer and Event Instructions modules, as it contains a single dictionary, *sequences*, that connects to a specific Event Instructions module when given the name of a sequence.

6.3.3.2 Event Instructions

Each Event Instructions module possesses a static function for returning an array of instruction arrays. This 2D array is eventually passed to the Sequencer to be parsed and turned into an on-screen cutscene.

6.3.3.3 Main Scenes

The Main Scenes resource script holds two dictionaries, *explore_scenes* and *battle_scenes*, that are used by the Scene Manager for changing between different scenes within the game.

6.3.3.x Enemy Types

The simple Enemy Types module hangs onto a single dictionary data structure, *enemy_types.* This dictionary stores the file paths for each enemy type's scene for instancing, as well as basic battle stats. The module frequently passes this dictionary up to the Enemy Handler so that enemies might be instanced in exploration mode.

6.3.3.5 Entity Stats

The Entity Stats module has an array named *stat_keys* that holds all of the character and enemy stat names. The *stats_template* dictionary turns each of these stat names into keys, and assigns each a default value of one, to provide a sample for how player and enemy stats should be structured and passed around to battle scenes.

6.3.3.6 Save Manager

The Save Manager implements an array called *save_files* to keep track of the names of all the possible save files in *Cosmonarium*. Each position in *save_files* contains a String value referring to one of these names. In addition to this array, the Save Manager manipulates files when storing persistent data on a given user's machine. These files may or may not be encrypted, depending on what kind of save is associated with the file itself.

6.3.3.7 Persistent Data

The Persistent Data module keeps track of all the data owned by persistent nodes in its *data* dictionary. This dictionary stores persistent data as subdictionaries, where each subdictionary, where each of these subdictionaries contains the true data to be saved, e.g. a player character's current level and stats, whether a boss enemy is still alive, etc The whole *data* dictionary is written to an actual persistent file by the Save Manager whenever a user successfully saves his or her game progress.

6.3.3.8 Background Audio Engine

The Background Audio Engine does not have any data structures of its own that it uses, as it mainly follows the directions of the Scene Manager and Sequencer for its behavior.

6.3.3.9 Camera Manager

The Camera Manager module does not boast any data structures of its own, as most of its functionality can be adequately handled by carefully manipulating Vector2 positions and Godot's Tween nodes.

6.3.3.10 Scene Manager

The Scene Manager module itself does not boast any major data structures. Instead, this module pulls from the Main Scenes module's data structures to organize which scenes are swapped in and out at any point during the game's execution.

6.3.3.11 Party

The Party module generally keeps track of three separate arrays. The *party* array very intuitively contains the names of those player characters currently within the party. The *incapacitated* array tracks which player characters are still alive, and which have been temporarily due to the results of a previous battle. Finally, the *items* array monitors the party's current inventory. Many of the consumable items belong to the player characters collectively, and so these items would be stashed in *items* before they have been used.

6.3.3.12 Boss Enemy

As a somewhat stripped down version of the Enemy Parent script, the Boss Enemy module has similar data structures to its associate. The *skins* dictionary provides all of the possible boss enemy skins that may be displayed, with each boss essentially having its own assigned skin. Each boss enemy also has a copy of the *dir_anims* dictionary, which contains keys and their associated arrays of animation names for determining which directions an instanced boss enemy may move and face.

6.3.3.13 Input Engine

The Input Engine module possesses a multitude of dictionaries and arrays for accepting and interpreting the user input provided via external interfaces. Four of these dictionaries are quite similar: *to_player_commands, to_menu_commands, to_dialogue_commands,* and *to_battle_commands.* Each of these dictionaries are used to translate the input key or button presses into methods to be called. As such, each method call is mapped to a specific button and button press type (pressed, just pressed, or just released). The *valid_receivers* dictionary specifies which components of the game can receive user input. Each of these receivers is then associated with a subdictionary that provides data about its priority in accepting input, which process loop it works with, and which of the four previously described translator dictionaries it uses for method call lookup. The *curr_input_receivers* array relates which of the valid input receivers are currently available, based on their priority. Finally, the *disabled* array contains any input receivers that may be temporarily disabled, such as the "Player" receiver during sequenced events.

6.3.3.14 Sequencer
The Sequencer stores event data in the *active_event* dictionary. Some of this data includes the actual instructions taken from the Event Instructions module via the Event Sequences module, and each individually parsed instruction for the supplied sequenced event is stored one by one in *current_instruction*.

6.3.3.15 Actor Engine
The Actor Engine uses three main data structures for working with and manipulating actors within the game environment. First, the module has a general *actors_array* that collects all of the existing actors within the current scene. These actor nodes are also stashed in the *actors_dict*, using the actor names as keys and the actual bodies (KinematicBody2D, RigidBody2D, etc.) as the values. The Actor Engine also has the *asynchronous_actors_dict* for keeping track of all the actors performing async commands. Async actors are added to and removed from this dictionary frequently during sequenced events.

6.3.3.16 Enemy Handler
The Enemy Handler singleton owns three main data structures that it uses in frequent operations. The first of these is the *existing_enemy_data* dictionary, which stores an enemies battle stats as a value of its *data_id* integer key. The Enemy Handler also has two arrays that work in tandem with one another. The first is *queued_battle_enemies*, which acts somewhat like a naive queue: enemy IDs are appended to the array and used (in order) to instance enemy AI players and their battle sprites in combat mode. Moving the opposite direction from combat mode to exploration mode, the IDs of defeated enemies are appended to *queued_despawns*, which allows the Enemy Handler to delete these enemies in the overworld.

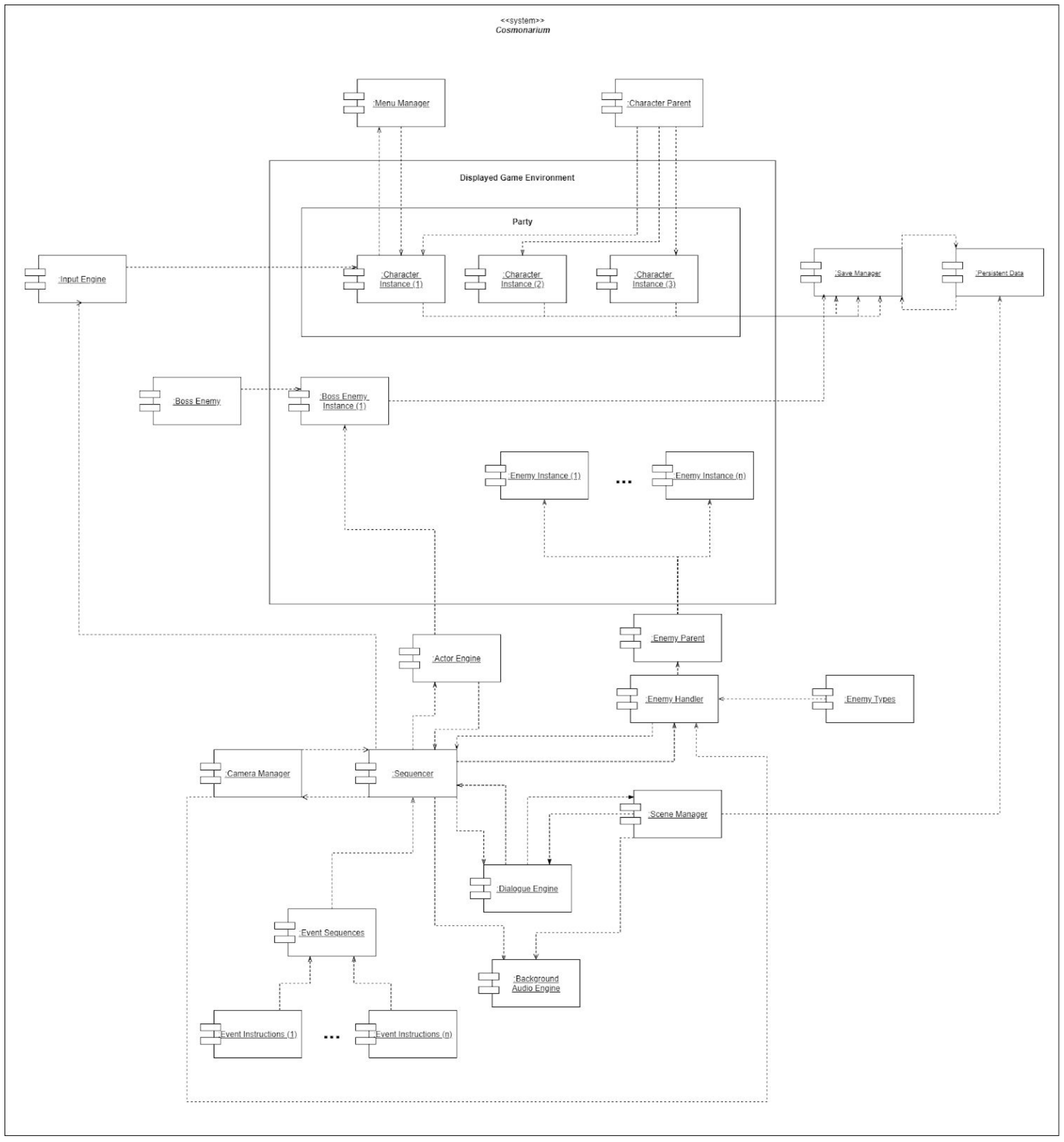
6.3.3.17 Enemy Parent
The Enemy Parent module hosts a variety of data structure implementations. The *skins* dictionary provides all of the possible enemy skins that may be displayed, with each enemy type essentially having its own assigned skin. Each enemy also holds onto a copy of the *patrol_patterns* array, which lists all of the possible ways an enemy might patrol around the overworld. Lastly, the *dir_anims* dictionary contains keys and their associated arrays of animation names for determining which directions an instanced enemy may move and face.

6.3.3.18 Character Parent
The Character Parent module hosts a variety of data structure implementations. The *skins* dictionary provides all of the possible character skins that may be displayed, e.g. whether a player character is wearing a coat or not. This dictionary of shared sprites simplifies how instanced characters can swap their exterior appearances. The *interact_areas* array collects all of the possible objects within the current scene with which the active player can interact. These objects may include non-player characters, treasure chests, scene props, and so on. The

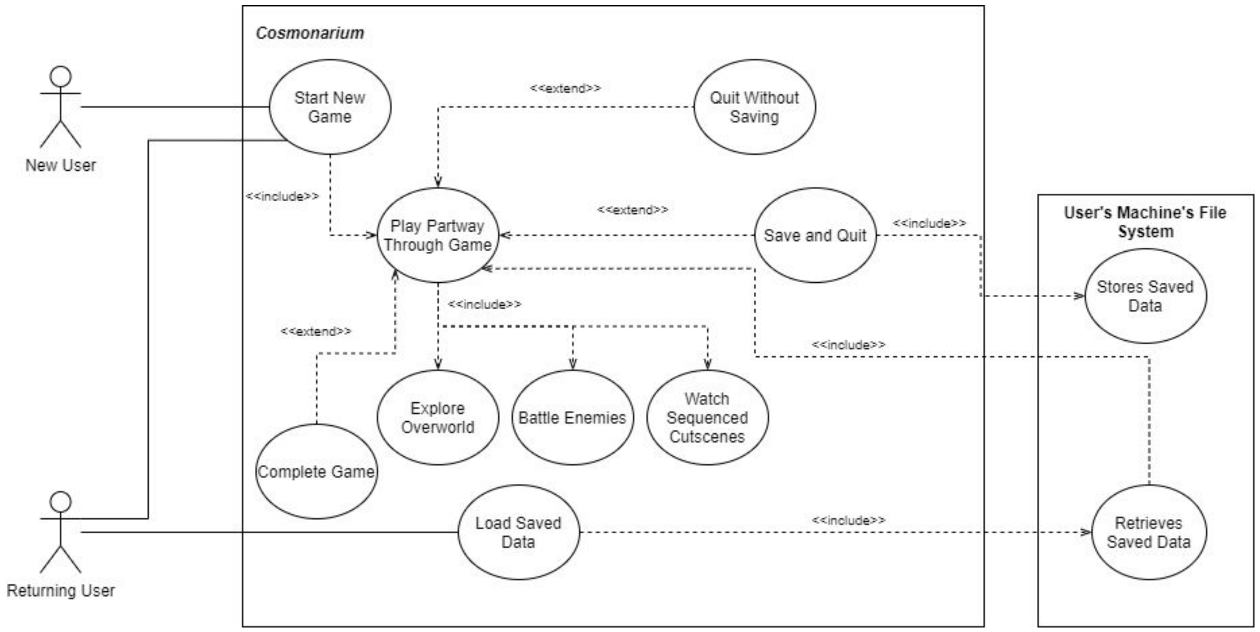
party_data dictionary that each player character possesses keeps track of any party-related information, such as whether the player character is currently following or not, or if a given character is currently the active player. Lastly, the *dir_anims* dictionary contains keys and their associated arrays of animation names for determining which directions a player character may move and face.

6.3.4 Detailed Design Diagrams

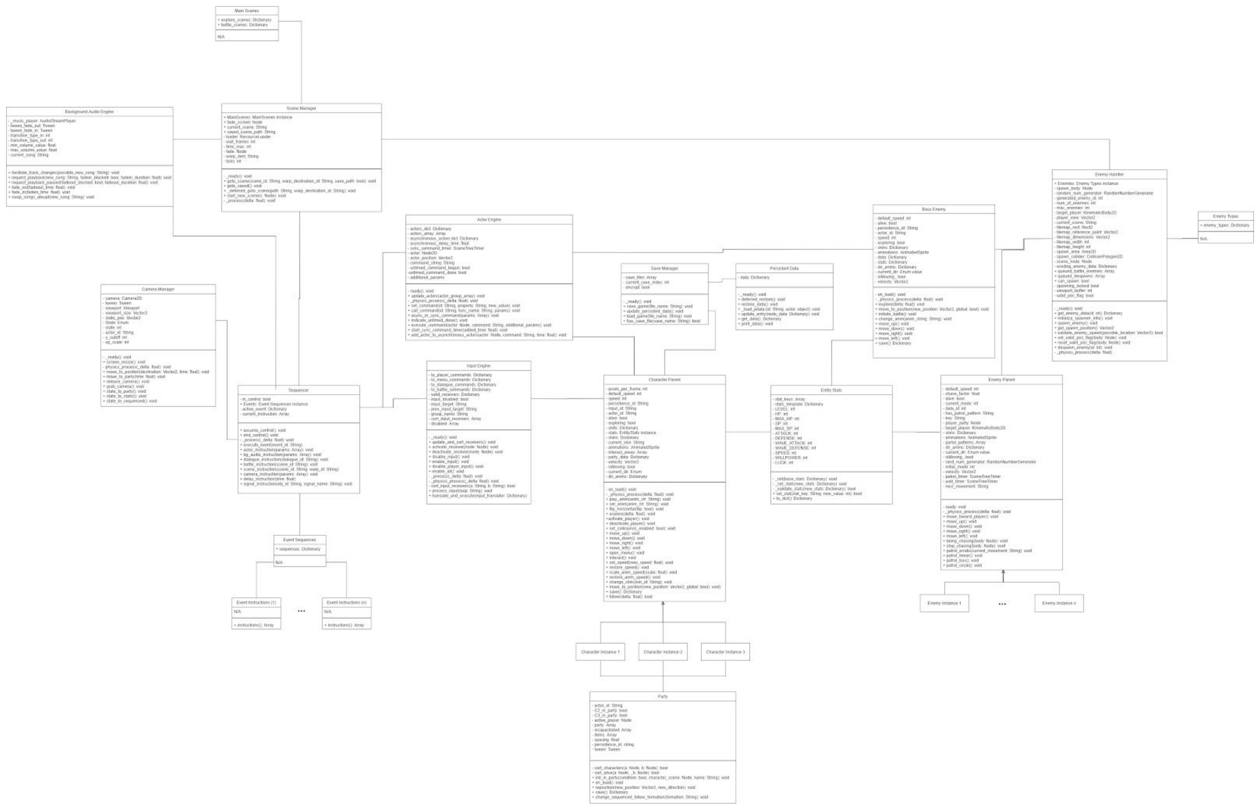
6.3.4.1 Detailed Component Diagram



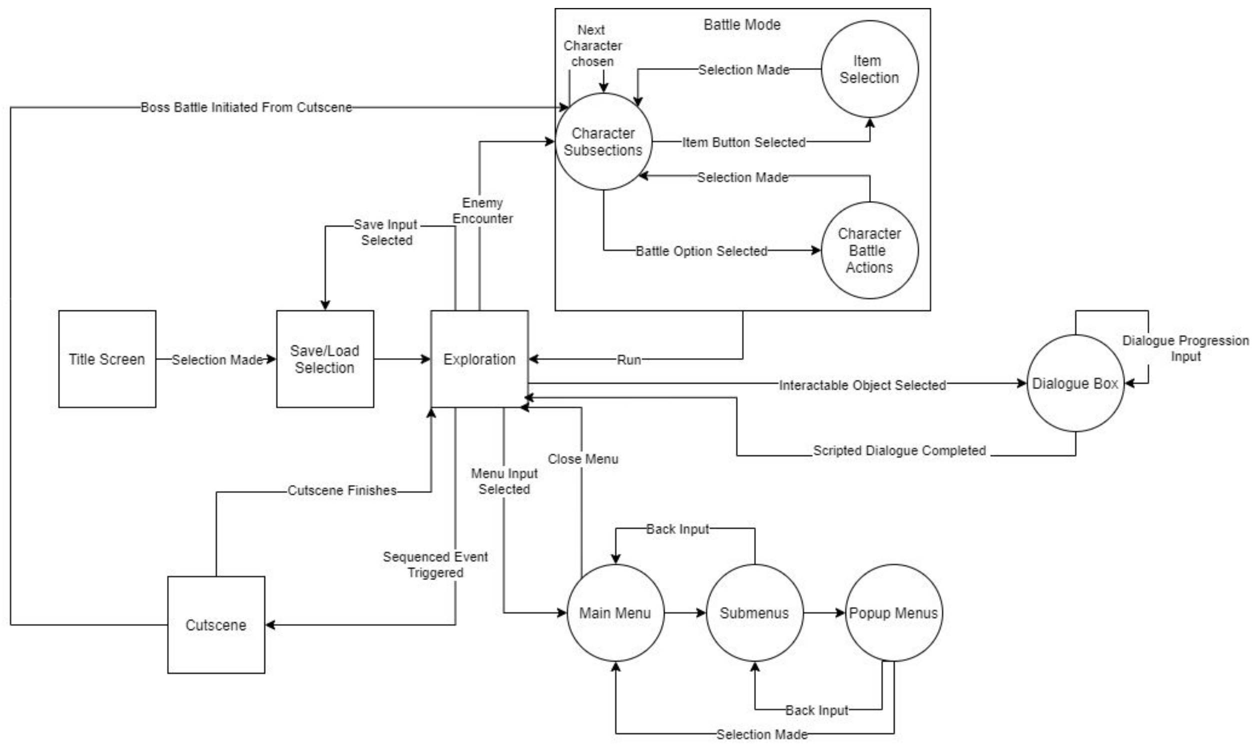
6.3.4.2 Detailed Use Case Diagram



6.3.4.3 Detailed Class Diagram (for Project's Modules)



6.3.4.4 Detailed State Diagram



Battle Mode
Next Character chosen
Item Selection
Selection Made
Boss Battle Initiated From Cutscene
Character Subsections
Item Button Selected
Selection Made
Enemy Encounter
Save Input Selected
Battle Option Selected
Character Battle Actions
Title Screen
Selection Made
Save/Load Selection
Exploration
Run
Dialogue Progression Input
Interactable Object Selected
Dialogue Box
Scripted Dialogue Completed
Close Menu
Menu Input Selected
Cutscene Finishes
Sequenced Event Triggered
Back Input
Cutscene
Main Menu
Submenus
Popup Menus
Back Input
Selection Made