DOCUMENTATION FOR THE FALL WITCHES ENGINE V0.1

Oleo-de-MACACO/**Fall-Witches-Project**

Fall Witches game project

1 Contributor | 0 Issues | 0 Stars | 0 Forks

TABLE OF CONTENTS:

OVERVIEW:

The "Fall Witches" engine is a FOSS, 2D, data-driven game engine developed in pure C with the Raylib library. It is designed with a modular architecture, where each component is responsible for a specific part of the game. This philosophy emphasizes a clean separation of responsibilities; each module (a .c file) provides a public API through its corresponding header (.h) file, while the core game logic is frequently controlled by external data files like .txt and .png files.

The main application loop and the orchestration of all modules are handled in `src/main.c`. The project is currently in an experimental state and is initially targeted for Linux systems (Debian, Ubuntu, and derivatives), with future plans for a Windows release.

The core modules of the engine include:

Game Engine (Game): Manages primary game states, the camera, and the main update logic.

Entities and Classes (Classes, CharacterManager): Defines the data structures and manages the behavior of players, NPCs, and enemies.
World System (WorldLoading, WorldMap, MapData): Handles the loading, rendering, and transitioning between map sections.
User Interface (Menu, PauseMenu, Inventory): A collection of modules that control the various UI screens.
Data Systems (SaveLoad, GameProgress): Manages the persistence of game data, including saves and player progress.
Scripting Systems (Event, Dialogue): Enables the creation of complex events and dialogues through external text files.
Audio System (Sound): Manages the playback of all music and sound effects.

WARNING: THIS ENGINE FOR NOW IS BEING DEVELOPED AS LINUX FIRST! EXPECT SUPPORT FOR WINDOWS BUILDS TO BE SLOWER!

DIALOGUE ENGINE:

The dialogue system is driven by an external text file, allowing for easy creation and modification of in-game conversations.

How it Works:

Definition File: All dialogues are defined in assets/Dialogues/dialogues.txt.

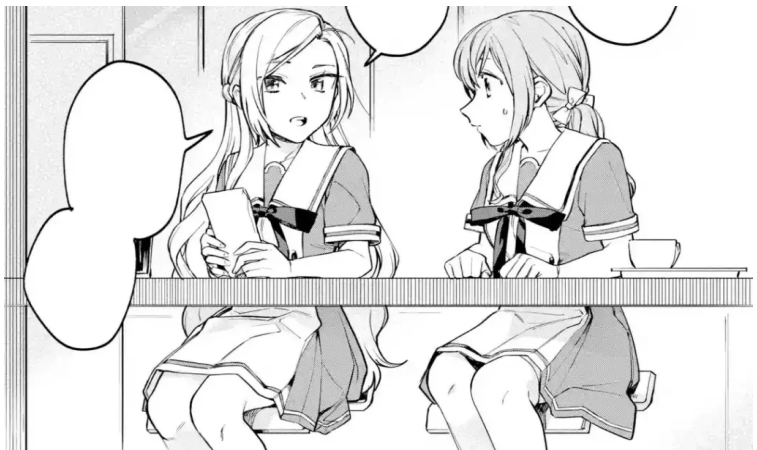
Format: A new dialogue sequence is identified by a line starting with # followed by a unique numeric ID. Each subsequent line represents a single piece of dialogue in the format Speaker: Text. Visual separators (---) can be used in the file but are ignored by the parser. If a line has no speaker name before the colon, the name will not be displayed in the UI.

Update Loop: When a dialogue is active (Dialogue_IsActive() returns true), it takes priority over the main game loop, pausing other game logic. The Dialogue_Update function manages the state and presentation of the dialogue.

"Typewriter" Effect: The system implements a visual "typewriter" effect where text appears character by character. A timer, s_typingTimer, controls the speed of the text reveal. The TIME_PER_CHAR constant determines the delay for each character.

Input Handling: The dialogue system uses a single action key (Enter or Mouse Click) for multiple functions. If the text is still being "typed," a key press will instantly reveal the entire line. If the line is already complete, the same key press advances to the next line of dialogue or ends the conversation if it's the final line.

Rendering: The Dialogue_Draw function uses Raylib's TextSubtext to get the currently visible portion of the text, which is then drawn to the screen to create the typing effect.



Example Dialogue File (dialogues.txt)

```
# 101 // Dialogue ID used by the Event system

Mysterious Figure: Greetings, traveler... You seem to be lost.

Hero: Who are you? Where am I?

: The ground trembles beneath your feet.

---

# 102 // A second, separate dialogue

Shopkeeper: Quality goods, straight from the forge!
```

SOUND ENGINE:

The engine features a robust audio manager that handles both music and sound effects (SFX) with categorized volume control.

How it Works:

Categorization: All audio is organized into categories defined by the MusicCategory enum in include/Sound.h. Categories include MUSIC_CATEGORY_MAINMENU, MUSIC_CATEGORY_GAME, MUSIC_CATEGORY_BATTLE, and various ambient categories like MUSIC_CATEGORY_AMBIENT_NATURE.

Automatic Loading: During initialization, the LoadGameAudio function automatically loads all supported audio files (.ogg, .mp3, .wav, .flac) from the corresponding subfolders within assets/audio/.
Volume Control: The system features separate volume controls for each category, in addition to a master volume slider. These can all be adjusted in the Settings screen. The final effective volume for any sound is calculated as categoryVolume * masterVolume. Changes made in the settings are applied in real-time for immediate user feedback.
Playback Management: The module keeps track of the currently playing music track, allowing it to be paused, resumed, or stopped entirely. For streaming music files, the UpdateAudioStreams function must be called every frame in the main loop to buffer the data.

EVENT ENGINE:

The event system is a powerful scripting tool that uses a simple text file to define complex, trigger-based in-game events.

How it Works

Definition File: All game events are defined in `assets/Events/events.txt`.

Event Structure: Each event begins with `event:` followed by a unique ID. The event's properties are enclosed in curly braces `{}`.

Triggers: An event is fired when its trigger condition is met. The primary trigger is `triggers: player_first_time_in_map=true`, which activates an event the first time a player enters a specific map section. This is tracked using the `GameProgress` module.

Actions: When triggered, an event executes a sequence of actions defined in an `actions:` block. Supported actions include `play_music:` and `show_dialogue:`, which start a music track or initiate a dialogue sequence, respectively.

Execution Flow: The `Event_CheckAndRun` function is called every frame. It iterates through all loaded events, checking if an event's location matches the player's current map coordinates. If it matches, it evaluates the trigger condition (e.g., by calling `Progress_HasVisitedMap`). If the trigger is met, the event's actions are executed, its `hasFired` flag is set, and `Progress_MapWasVisited` is called to update the player's progress.

Example Event File (events.txt):

```
// Defines a new event with ID 1

event: 1 {

// This event can only occur on map (0,0)

map = (0,0)

// This event triggers only the first time the player enters map (0,0)

triggers: player_first_time_in_map=true

// List of actions to execute when triggered

actions: {

 // Action 1: Play a music track

play_music: { category = Game songname = your_song_name.ogg loop = true }

// Action 2: Start a dialogue sequence

show_dialogue: {

// This ID must correspond to an ID in dialogues.txt

id = 101 } } }
```

MAP-SPECIFIC VARIABLES ENGINE:

This system, parsed by src/MapData.c, allows for fine-tuning the properties of a specific map section through an optional text file.

How it Works:

File Location: The engine looks for a file in `assets/MapVariables/` with the name `X-Y.txt`, where X and Y are the map coordinates.

Parsing: The parser ignores commented lines (starting with #) and reads properties in two ways: simple key-value pairs and multi-line blocks.
Properties & Blocks:

`enemy_spawn_chance: [value]`: A general property setting the percentage chance of an enemy spawning at a set interval.

`allowed_music_start` / `allowed_music_end`: A block that defines a list of music categories that can be played randomly on this map.
`npc_start` / `npc_end`: A block that defines a static NPC for the map, including its name, dialogue ID, sprite folder, and spawn coordinates.
`enemy_type_start` / `enemy_type_end`: A block that defines a type of enemy that can spawn on this map, including its name and sprite folder.

Example Variables File (0-0.txt):

```
# Configuration for the starting forest map (0,0)

# 30% chance for an enemy to appear at each spawn interval

enemy_spawn_chance: 30

# Ambient music categories that can play here
allowed_music_start

Nature

Game

allowed_music_end

# ---- Defines a static NPC ----

npc_start name: Old Hermit

dialogue_id: 101

sprite_folder: npc/old_man

spawn_coords: 450.0, 320.0

npc_end

# ---- Defines an enemy type that can spawn on this map ----

enemy_type_start

name: Forest Slime

sprite_folder: enemies/slime

enemy_type_end
```

MAP LOADING ENGINE:

The world is a large grid, and the engine loads the environment in sections using an innovative system that combines a visual image with a data image.

How it Works:

File Structure: Each map section `(X,Y)` is defined by up to three files:

1. `assets/WorldTextures/section_X_Y.png`: The visual texture of the map. Its dimensions dictate the size of the map section.

`assets/WorldPlaces/section_X_Y.png`: A data map where specific colors represent game logic elements. This file **must** have the same dimensions as the texture file.
`assets/MapVariables/X-Y.txt`: An optional file for additional map-specific data (see previous section).

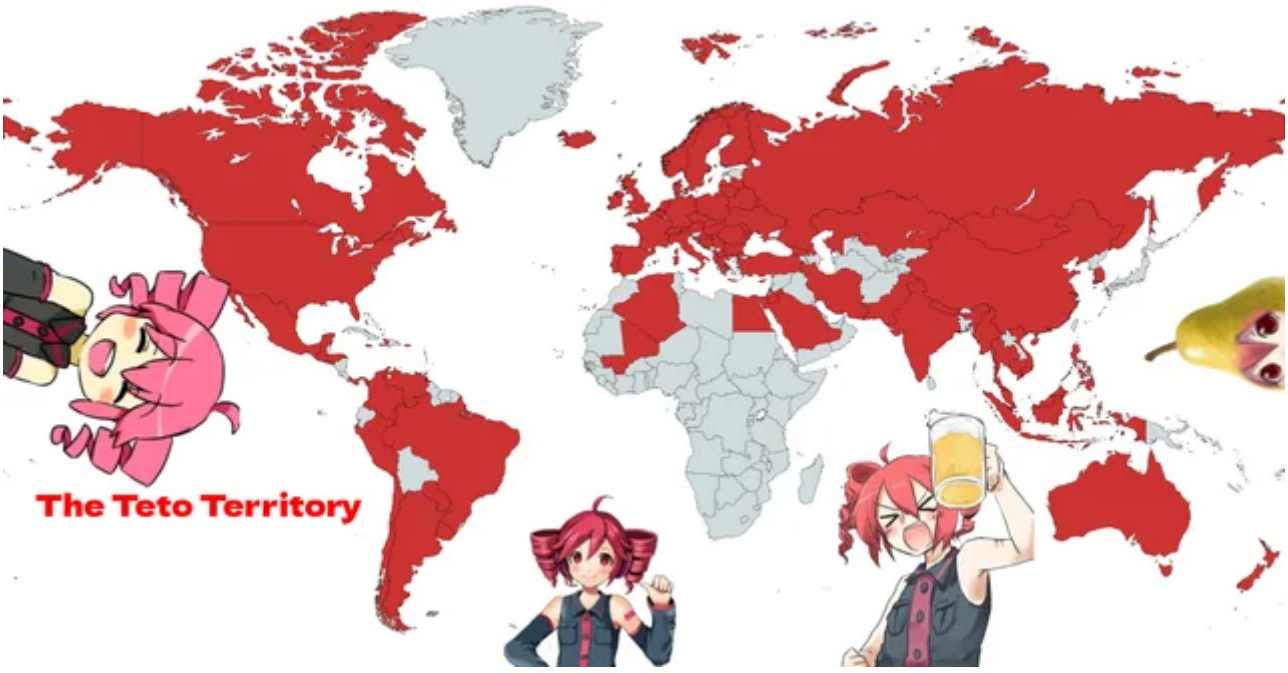
3.

Loading Process (`LoadWorldSection`): When loading a map, the engine first loads the background texture. It then loads the corresponding "places" image and validates that their dimensions match. Finally, it iterates through every pixel of the "places" image and, based on the color, generates game objects like collision rectangles.
Color-Coded Data: The colors in the `WorldPlaces` image correspond to specific game objects, as defined in `include/WorldLoading.h`:

Red (`COLOR_COLLISION`): Defines a collision tile that characters cannot pass through. Movement is handled by the `move_character` function, which checks for collisions against these generated rectangles.

Blue (`COLOR_DOOR`): Marks the location of a door.

Green (`COLOR_PLAYER_SPAWN`**)**: Defines a potential spawn point for a player.
Magenta (`COLOR_ENEMY_SPAWN`**)**: Marks a spawn point for an enemy.
Yellow (`COLOR_CHEST_SPAWN`**)**: Indicates the location of a chest.
Cyan (`COLOR_FORAGE_SPAWN`**)**: Marks a spot where collectible items can be found.
Map Transitions (`WorldMap_CheckTransition`**):** This function checks if a player has reached the edge of the current map section. If so, it calculates the new map coordinates, validates them against the world boundaries, updates the global map coordinates, and repositions the player on the opposite side of the new map section to ensure a seamless transition. For two players, the transition only occurs if both players are on the same border.



SAVE SYSTEM:

The engine uses a binary file format to save and load game progress, with built-in version control and error checking to ensure data integrity.

HOW IT WORKS:

- **File Format & Location: Game saves are stored as binary .sav files. They are located in separate directories based on the game mode: Saves/SinglePlayer/ or Saves/TwoPlayer/.**

  Version Control: A constant, `SAVEGAME_FILE_VERSION`, is defined in `src/SaveLoad.c`. This version number is the very first piece of data written to a save file. When loading, the engine compares the file's version with its own; if they don't match, the load is aborted to prevent corruption from incompatible save structures.

  Error Checking: The system uses `FWRITE_CHECK` and `FREAD_CHECK` macros to wrap all file I/O operations. These macros verify that the number of bytes written or read matches the expected amount. If a discrepancy occurs, they log a detailed error message and safely terminate the operation, preventing a corrupted save.

  Saved Data: The system saves data in a strict, specific order:

  Save File Version;
  Current Map X and Y coordinates;
  Number of active players;
  The entire `g_gameProgress` struct (which includes data like visited maps);

  A loop for each player, saving every field of their `Player` struct in sequence.

  Game Progress: The `GameProgress` system is crucial for persistent data that should be part of the save file, such as tracking which maps have been visited for the event system. When a new game is started by overwriting an existing slot, `Progress_Reset()` is called to clear this data.

SPRITE LOADING SYSTEM:

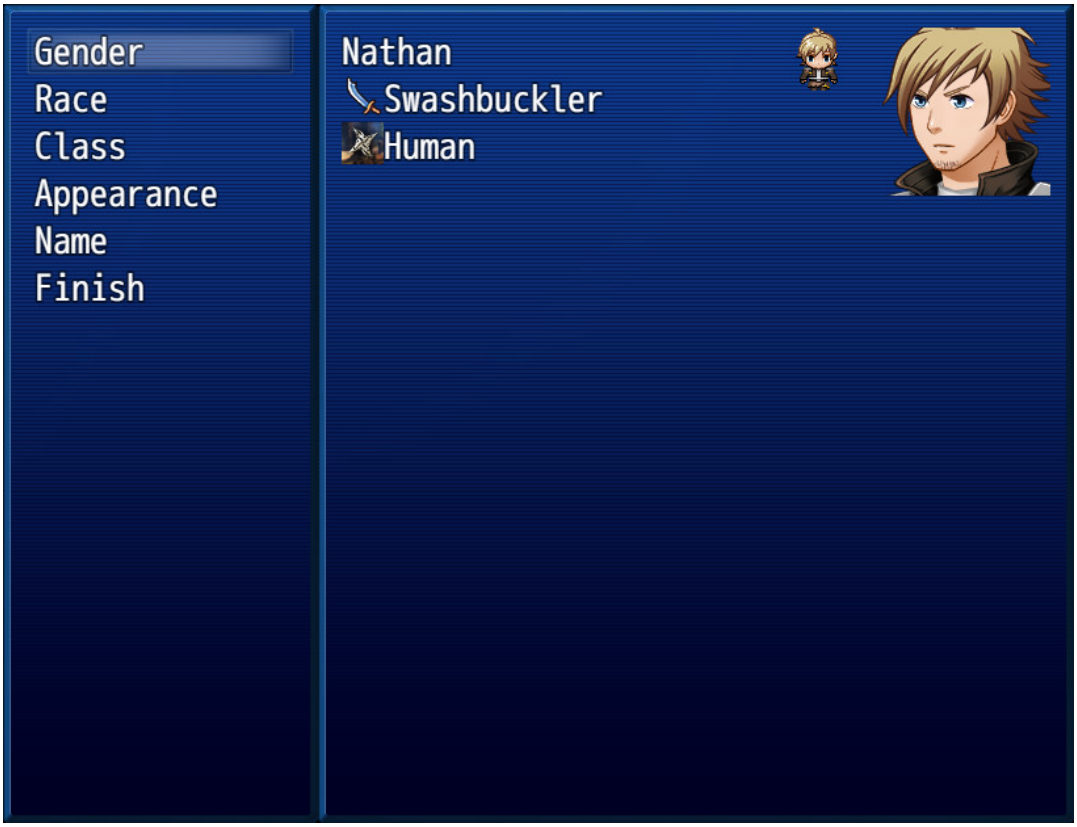
This system handles the loading and management of character animations.

How it Works:

`LoadCharacterAnimations`: This function, found in `src/WalkCycle.c`, is responsible for loading all the animation frames for a character. It determines which set of sprites to load based on the player's `spriteType` (e.g., `SPRITE_TYPE_HUMANO`). The sprites are expected to be in a specific folder structure, such as `assets/characters/[sprite_folder]/walk_up_0.png`.

`UpdateWalkCycle`: This function is called every frame to update a character's animation state. It checks if the character is moving and in which direction, and then advances `currentAnimFrame` based on a timer (`frameTimer`) to cycle through the appropriate animation frames.
`GetCurrentCharacterSprite`: This function returns the correct texture that should be drawn for the character in the current frame, based on their movement status, direction, and animation frame index.

CHARACTER CREATION:



This screen allows the user to define their character at the start of a new game. The process involves several steps managed by the CharacterCreation module:

1. **Enter Name:** The player types a name for their character.
2. **Choose Class:** The player selects from the available classes (`GUERREIRO`, `MAGO`, etc.), which determines their base stats.
3. **Choose Race/Sprite:** The player selects a `SpriteType` (e.g., `Humano`, `Demonio`), which determines the character's visual appearance and which set of sprites will be loaded.

S.P.E.C.I.A.L STATS:

As part of the classic JRPG-style design, each character possesses a set of core attributes inspired by the S.P.E.C.I.A.L. system. These attributes, defined in the Player struct in include/Classes.h, are:

- `forca` (Strength)
- `percepcao` (Perception)
- `resistencia` (Endurance)
- `carisma` (Charisma)
- `inteligencia` (Intelligence)
- `agilidade` (Agility)
- `sorte` (Luck)

These stats are assigned default values based on the character's class during initialization.

OTHER BITS & BOBS:

Main Game Loop

The heart of the application is the `while` loop in `src/main.c`. It continuously runs until the window is closed or the `g_request_exit` flag is set. Inside this loop, a large `switch` statement based on the `currentScreen` variable calls the appropriate `Update...` function for the current game state (e.g., `UpdateMenuScreen`, `UpdatePlayingScreen`).

Collision Detection (move_character):

To prevent characters from "tunneling" through thin walls at high speeds, the `move_character` function in `src/Game.c` moves the character in 1-pixel increments. For each pixel of movement, it checks for a collision. If a a collision is found, the move is undone, and the character stops, effectively sliding along the wall.

Rendering Pipeline

The engine uses a virtual resolution of 800x450 to ensure consistent pixel art scaling, regardless of the actual window size. The rendering process happens in distinct layers:

Begin Rendering: `BeginDrawing()` is called.

Redirect to Texture: `BeginTextureMode(targetRenderTexture)` redirects all subsequent draw calls to an off-screen render texture instead of the main window buffer.
Draw Game World: If in a game state that uses the camera, `BeginMode2D(gameCamera)` is called. The world background, characters, and players are then drawn.
Draw Game UI: UI elements that exist in the game world, like a pause menu, are drawn on top, still within the `TextureMode`.
Restore Rendering: `EndTextureMode()` restores the render target back to the main window.

Draw Final Texture: The entire `targetRenderTexture` is drawn to the screen using `DrawTexturePro`, which scales it appropriately and adds

letterboxing to maintain the aspect ratio. A negative Y-coordinate is used on the source texture to correct for OpenGL's texture orientation.

Draw Overlay UI: Finally, elements that should appear over everything else at native resolution, like the dialogue box, are drawn after the main texture has been rendered.

End Frame: `EndDrawing()` presents the final composite image to the user.

UI System:

Most UI interactions are powered by the MenuButton struct, defined in include/Menu.h. This struct holds a Rectangle for positioning and collision, text, colors for various states (normal, hover, disabled), and an action enum that determines what happens on a click. The Update function for each UI screen checks for mouse collision with these button rectangles and executes the corresponding action when a click is detected.

LEVELING SYSTEM:

The engine includes a simple yet effective leveling system to handle character progression.

- **How it Works:**
    - **XP Trigger:** A player is eligible to level up once their `exp` attribute reaches or exceeds 100.
    - **Leveling Up:** The `LevelUpPlayer` function in `src/Classes.c` is called automatically after a victorious battle where XP is awarded.
    - **Process:**
        1. The player's `nivel` is incremented by one.
        2. 100 XP is subtracted from the player's `exp` total, allowing any excess XP to carry over to the next level.
        3. The player's core stats (`max_vida`, `ataque`, `defesa`, etc.) are increased by a set amount.
        4. **Class-Specific Stats:** The stat gains are tailored to the character's class. For example, a `GUERREIRO` gains more `forca` and `max_vida`, while a `MAGO` gains more `inteligencia` and `max_mana`.
        5. The player's `vida` and `mana` are fully restored to their new maximum values.

BATTLE SYSTEM:

The engine features a classic, turn-based battle system driven by character stats and external data files. The logic is primarily handled by `src/BattleSystem.c` and `src/BattleUI.c`.

- **How it Works:**
  - **Initiation:** Battles are triggered by the `CharManager_TriggerBattle` function when a player collides with an enemy on the map.
  - **Turn-Based Flow:** The battle follows a strict sequence managed by the `BattlePhase` enum (`BATTLE_STATE_PLAYER_TURN`, `BATTLE_STATE_ENEMY_TURN`, etc.). A timer (`ACTION_DELAY`) ensures a pause between actions for readability.
  - **Player Actions:** On the player's turn, the UI allows them to select an action. The "Habilidades" command lets the player choose from a list of attacks loaded from `ClassAttacks.txt`.
  - **Action Processing:** Player actions and enemy AI attacks are resolved in `ProcessPlayerAction` and `ProcessEnemyAction`, respectively. Damage calculations use character stats, attack power multipliers, and a D20 roll for a degree of randomness.
  - **Conclusion:** The battle ends when all members of one side have their HP reduced to zero. The `BattleSystem_End` function handles the conclusion, applying results like HP/MP changes and awarding XP and coins on victory.