

Project Report: The Way of the Shadow

A 2D Platformer Game
Developed using Pygame

Adithyan Karanathu Shibu
202307583

Course: CSCI-485.20

Abstract

This report details the design, development, and implementation of "The way of the Shadow", a 2D-platformer game created using the Pygame library for Python. The project aimed to create an engaging single-player experience featuring core platforming mechanics such as precise movement, jumping, double jumping, wall sliding, and wall jumping. Key features include a multi-level structure loaded from configuration files, a collectible system ("Scroll Coins"), a timed high-score system with persistent saving/loading, and a dynamic power-up mechanic granting temporary speed and jump boosts upon collecting a specific number of scrolls. The game architecture follows a modular design, separating concerns into distinct Python modules for settings, sprites, levels, UI elements, and the main game logic. The development process involved iterative implementation of features, asset integration (graphics and audio), and debugging, including resolving challenges related to collision detection and physics tuning, particularly for advanced mechanics like wall interactions and the power-up system. The final product successfully demonstrates the application of Pygame for 2D game development, object-oriented principles, state management, and incorporates several engaging gameplay features.

Snapshot of the title screen

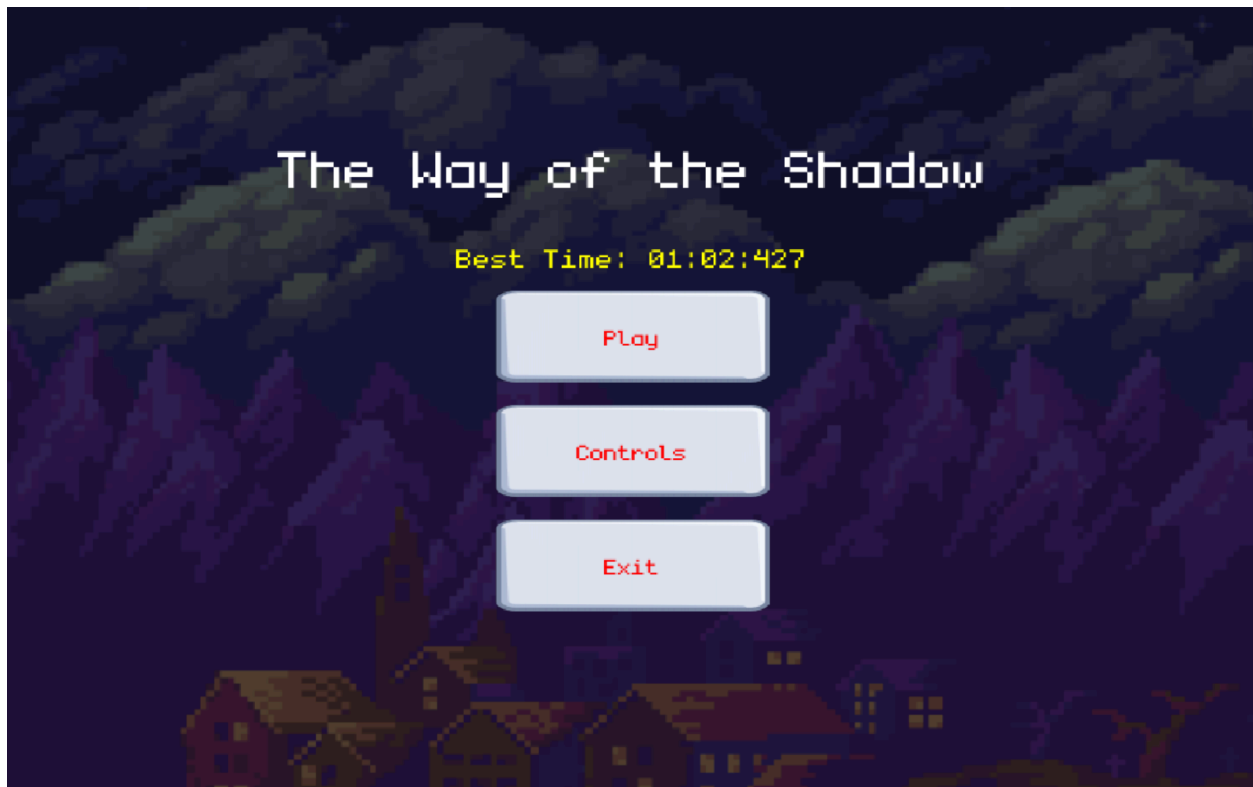


Table of Contents

Introduction - 4

Design & Architecture - 7

Core Gameplay Mechanics Implementation - 11

Feature Implementation - 15

Asset Management - 23

Testing and Debugging - 25

Results and Discussion - 28

Future Work and Improvements - 31

Conclusion / Final Thoughts - 32

Bibliography / References - 33

Appendices - 34

1. Introduction

1.1 Project Overview and Purpose

"The Way of the Shadow" is a single-player, 2D side-scrolling platformer game developed using the Python programming language and the Pygame library. The game challenges the player to navigate a series of levels by running, jumping, and utilizing advanced movement techniques like wall sliding and wall jumping. The primary objective within each level is to collect scattered "Scroll Coins" and reach a designated goal point as quickly as possible.

The purpose of this project was multifaceted:

- To gain practical experience in 2D game development principles and techniques.
- To demonstrate proficiency in using the Pygame library for handling graphics, input, sound, and game logic.
- To apply object-oriented programming (OOP) concepts in structuring a game application.
- To implement core and advanced platforming mechanics found in classic and modern games of the genre.
- To create a playable and engaging game prototype incorporating features like level progression, scoring, timing, and power-ups.
- The game draws inspiration from classic platformers, emphasizing fluid movement, level exploration, and a time-based challenge.



1.2 Project Goals and Objectives

The specific goals set for the "The way of the Shadow" project were:

- Core Mechanics: Implement robust and responsive player controls for walking, running, jumping, and double jumping.
- Advanced Mechanics: Successfully integrate wall sliding and wall jumping mechanics to enhance player mobility and level design possibilities.
- Level System: Design and implement a system for defining and loading multiple distinct game levels from an external configuration.
- Collectibles & Scoring: Incorporate collectible items (Scroll Coins) that contribute to a level score.
- Game State Management: Create a clear state machine to handle transitions between different game phases (e.g., Menu, Playing, Game Over, Level Complete, Game Won).
- Timer & High Score: Implement an accurate in-game timer tracking both individual level time and total game time. Develop a persistent high-score system to save the best completion time.
- Power-Up System: Design and code a power-up mechanic triggered by collecting a set number of scrolls, providing temporary enhancements to player speed and jump height, including an extension mechanism.
- User Interface (UI): Develop a functional UI including menus, buttons, a Heads-Up Display (HUD) showing score, level, timer, and power-up status.
- Audiovisuals: Integrate background music and sound effects for key actions (jump, collect) and basic sprite animations (idle, run).
- Modularity & Readability: Structure the codebase using Python modules and classes for better organization and maintainability.

1.3 Scope and Limitations

The scope of this project includes the development of the core game engine, player character, level elements (platforms, collectibles, goal), the features listed in the objectives (timer, high score, power-up, UI, audio, animation), and a small set of sample levels demonstrating these features.

The project has the following limitations:

- No Enemies: The current version does not include enemy characters or combat mechanics.
- Limited Level Count: Only a few predefined levels are included for demonstration. No level editor is provided.

- Basic Graphics: Assets used are functional but may not be highly polished artistically. Focus was on mechanics over aesthetics.
- Basic Animations: Player animations are limited to idle and run states. Jump, fall, and wall slide animations are not implemented visually (though the mechanics exist).
- Limited Sound Design: Audio integration is basic, covering background music and essential sound effects.
- Platform: Developed and tested primarily on a standard desktop environment. Compatibility with other platforms is not guaranteed without modification.

1.4 Technology Stack

The project was developed using the following technologies:

- Programming Language: Python 3.x
- Core Library: Pygame (version 2.x recommended) - Used for graphics rendering, event handling, audio playback, sprite management, vector math, and timing.
- Code Editor/IDE: PyCharm.

1.5 Report Structure

This report is organized into the following sections:

Section 1: Introduction: Provides the project context, goals, scope, and technology.

Section 2: Design & Architecture: Discusses the overall structure, game states, game loop, file organization, and class design.

Section 3: Core Gameplay Mechanics Implementation: Details the implementation of player movement, jumping, and wall mechanics.

Section 4: Feature Implementation: Covers the development of specific systems like levels, scoring, timer, high score, power-ups, UI, animation, and audio.

Section 5: Asset Management: Describes the handling of graphical and audio assets.

Section 6: Testing & Debugging: Outlines the testing process and key challenges faced.

Section 7: Results & Discussion: Evaluates the project against its objectives and discusses the outcome.

Section 8: Future Work & Improvements: Suggests potential enhancements and future directions.

Section 9: Conclusion / Final Thoughts: Summarizes the project and learning outcomes.

Bibliography/references

Appendices: Includes selected code snippets and potentially other supplementary materials.

2. Design and Architecture

This section delves into the architectural decisions and design patterns employed in the development of "The way of the shadow". A modular approach was prioritized to enhance code organization, readability, and maintainability.

2.1 Overall Architecture

The game is built around a central Game class (main.py) that manages the main game loop, game states, timing, high scores, power-up status, asset loading, and level transitions. It orchestrates interactions between different game components. The architecture utilizes a state machine pattern to control the flow of the game, switching between different modes like the main menu, gameplay, controls display, and end screens. Object-oriented principles are applied extensively, with distinct classes representing game entities (Player, Platform, Collectible, Goal) and UI elements (Button).

2.2 Game States

A state machine manages the different phases of the game. Constants (defined in settings.py) represent each state, and the game.game_state variable tracks the current state. The main loop's events, update, and draw methods behave differently based on the current state.

The primary game states include:

- STATE_MENU: The initial state, displaying the title, high score, and buttons to play, view controls, or exit.
- STATE_CONTROLS: Displays instructions on how to play the game, including movement, actions, and the power-up mechanic.
- STATE_PLAYING: The main gameplay state where the player controls the character, interacts with the level, collects scrolls, and the timer runs.
- STATE_LEVEL_COMPLETE: Displayed when the player reaches the goal in a level (but not the final level). Shows level time and provides options to proceed or return to the menu.
- STATE_GAME_OVER: Activated when the player fails (e.g., falls off the screen). Provides options to restart the level or return to the menu.
- STATE_GAME_WON: Displayed after completing the final level. Shows the final total time, compares it with the high score, and offers a return to the menu.

2.3 Core Game Loop

The heart of the game resides in the `Game.run()` method. Each iteration of the loop performs three critical steps, managed based on the current `game_state`:

Event Handling (`Game.events()`): Processes all user input (keyboard presses, mouse clicks, window closing events). It checks for global inputs (like fullscreen toggle, quitting) and state-specific inputs (like button clicks in menus, player actions during gameplay, restarting). State transitions are often triggered within this phase.

Updating (`Game.update()`): Updates the game logic based on the time elapsed since the last frame (delta time or `dt`). This includes:

- Updating player physics (position, velocity, acceleration based on input and gravity).

- Handling collisions between game objects.

- Updating animations for sprites.

- Incrementing timers (level_elapsed_time).

- Checking for win/lose conditions.

- Managing the power-up timer and state.

- Updating sprite groups.

Drawing (`Game.draw()`): Renders the current state of the game to the screen. This involves:

- Drawing the background.

- Drawing all active sprites (platforms, collectibles, player, goal).

- Drawing UI elements (HUD text, buttons relevant to the current state).

- Applying overlays for end screens.

Finally, updating the display to show the newly drawn frame (`pygame.display.flip()`).

The loop's speed is regulated using `pygame.time.Clock().tick(FPS)` to maintain a consistent frame rate defined in `settings.py`.

2.4 File Structure

The project code is organized into multiple Python files (.py) to promote modularity:

Name	Date modified	Type	Size
.idea	01-04-2025 19:27	File folder	
__pycache__	02-04-2025 21:16	File folder	
assets	27-03-2025 20:43	File folder	
Game.py	02-04-2025 21:16	PY File	37 KB
highscore.txt	29-03-2025 20:55	Text Document	1 KB
levels.py	29-03-2025 18:20	PY File	13 KB
main.py	02-04-2025 21:17	PY File	1 KB
settings.py	02-04-2025 17:23	PY File	3 KB
sprites.py	02-04-2025 18:06	PY File	19 KB
ui.py	27-03-2025 20:50	PY File	3 KB

1. main.py: the main file. Game class initializes and runs a **Game** instance, then quits Pygame and exits the script.
2. Game.py: orchestrating the game loop, state management, event handling, high-level updates, drawing calls, asset loading initiation, and power-up state management. It acts as the central hub.
3. sprites.py: Defines the classes for all game objects that appear on screen: Player, Platform, Collectible, Goal. These classes encapsulate the behavior and data (image, rect, physics, animation) for each entity type.
4. settings.py: Stores global constants and configuration variables used throughout the project, such as screen dimensions, FPS, colors, player physics parameters (gravity, jump power, speed, friction), animation speeds, file paths for assets, game state constants, and power-up parameters. This centralizes configuration for easy tuning.
5. levels.py: Defines the structure and data for all game levels. It typically contains a dictionary or list (LEVELS) where each entry details the placement of platforms, collectibles, the goal, and the player's starting position for a specific level. MAX_LEVELS constant is also defined here.
6. ui.py: Contains helper functions and classes for user interface elements. This includes the Button class (handling appearance, hover effects, and clicks) and the draw_text function for rendering text on the screen.
7. assets/ (Directory): Contains subdirectories for game assets.
8. img/: Stores image files (spritesheets, background, tiles, UI elements).
9. snd/: Stores audio files (background music, sound effects).
10. font/: Stores font files (optional, if using custom fonts).
11. highscore.txt (File): A simple text file used to store the best completion time persistently between game sessions.

2.5 Class Design

Object-oriented programming was central to the game's design. Key classes include:

Game.py and main.py:

Responsibilities: Main loop control, state management, event polling, timekeeping (level/total/power-up), high score persistence, asset loading coordination, level loading/transitioning, sprite group management, global power-up state tracking.

Key Attributes: screen, clock, running, game_state, all_sprites, platforms, collectibles, player, current_level_index, level_elapsed_time, total_game_time, high_score, powerup_active, powerup_end_time, etc.

Key Methods: __init__, setup_game_variables, run, events, update, draw, load_level, load_assets, save_highscore, state-specific draw methods (draw_menu, draw_playing, etc.).

Player (sprites.py, inherits pygame.sprite.Sprite):

Responsibilities: Represents the player character, handles input-driven movement, physics simulation (gravity, acceleration, friction), jumping, double jumping, wall sliding, wall jumping, collision detection/response with platforms, managing animation state, holding visual image and collision rect.

Key Attributes: game (reference), image, rect, pos, vel, acc, on_ground, jumps_left, wall_sliding, wall_slide_side, animation frames lists (idle_frames_r, run_frames_l, etc.), current_frame_index, last_anim_update, facing_right, current_action.

Key Methods: __init__, load_images, _extract_frames, reset, update, jump, animate, check_collisions_x, check_collisions_y.

Platform (sprites.py, inherits pygame.sprite.Sprite):

Responsibilities: Represents static solid ground or walls. Primarily defined by its position and dimensions (rect). Holds its visual image (often tiled). Used for collision checks.

Key Attributes: game (reference), image, rect.

Key Methods: __init__. (No update needed for static platforms).

Collectible (sprites.py, inherits pygame.sprite.Sprite):

Responsibilities: Represents items the player can pick up (Scroll Coins). Manages its own animation. Removed upon collision with the player.

Key Attributes: frames, image, rect, current_frame_index, last_anim_update.

Key Methods: `__init__`, `update`.

Goal (sprites.py, inherits pygame.sprite.Sprite):

Responsibilities: Represents the level's end point. Triggers level completion or game win upon collision with the player.

Key Attributes: game (reference), image, rect.

Key Methods: `__init__`.

Button (ui.py):

Responsibilities: Represents clickable UI buttons. Handles drawing text/images, detecting mouse hover for visual feedback, and checking for click events.

Key Attributes: rect, text, font, normal_image, hover_image, is_hovering.

Key Methods: `__init__`, `draw`, `check_hover`, `is_clicked`.

3. Core Gameplay Mechanics Implementation

This section details the implementation specifics of the fundamental player actions and interactions that define the platforming experience in "The way of the shadow".

3.1 Player Movement Physics

Player movement aims for a responsive yet physics-based feel. It leverages 2D vectors (`pygame.math.Vector2`) for precise calculations of position (`pos`), velocity (`vel`), and acceleration (`acc`).

Input Handling: In `Player.update`, `pygame.key.get_pressed()` checks the state of movement keys (A/D or Left/Right arrows).

Acceleration: If a movement key is pressed, a horizontal acceleration (PLAYER_ACC from settings.py, potentially modified by the power-up speed_mult) is applied in the corresponding direction.

Gravity: A constant downward acceleration (PLAYER_GRAVITY) is applied vertically in every frame update ($\text{self.acc.y} = \text{PLAYER_GRAVITY}$).

Friction: When no horizontal movement keys are pressed, a friction force is applied to decelerate the player horizontally. This is implemented by adding an acceleration opposite to the current velocity ($\text{self.acc.x} += \text{self.vel.x} * \text{PLAYER_FRICTION}$, where PLAYER_FRICTION is a negative value, e.g., -0.1).

Velocity Update: Velocity is updated based on acceleration ($\text{self.vel} += \text{self.acc}$).

Speed Capping: Horizontal velocity (self.vel.x) is capped at a maximum value (MAX_RUN_SPEED, modified by speed_mult during power-up) to prevent excessive speeds. A minimum threshold check ($\text{abs}(\text{self.vel.x}) < 0.1$) sets velocity to zero to prevent minor drifting. Maximum fall speed (MAX_FALL_SPEED) is also applied to self.vel.y .

Position Update: The player's precise position (self.pos) is updated using the velocity and acceleration ($\text{self.pos} += \text{self.vel} + 0.5 * \text{self.acc}$). The player's collision rectangle (self.rect) position is then updated by rounding the precise vector position ($\text{self.rect.x} = \text{round}(\text{self.pos.x})$).

Collision Response: After updating the rect position, collision checks (check_collisions_x, check_collisions_y) are performed. If a collision occurs, the rect is snapped to the edge of the colliding platform, the corresponding velocity component is zeroed, and the pos vector is re-synchronized with the adjusted rect position.



3.2 Jumping and Double Jumping

The jumping mechanic allows the player to overcome obstacles and gain vertical height.

Initiation: The `Player.jump()` method is called when the jump key (Space, W, or Up Arrow) is pressed (detected in `Game.events`).

Ground Jump: If `self.on_ground` is `True`, a negative vertical velocity (`PLAYER_JUMP_POWER` from `settings.py`, modified by `jump_mult` during power-up) is applied to `self.vel.y`. The `on_ground` flag is set to `False`, and the `jumps_left` counter (initialized to 2) is decremented.

Double Jump (Air Jump): If `self.on_ground` is `False` but `self.jumps_left` is greater than 0, applying the jump key triggers the double jump. A potentially different negative vertical velocity (`PLAYER_DOUBLE_JUMP_POWER`, modified by `jump_mult`) is applied, and `jumps_left` is decremented again.

Jump Refresh: The `jumps_left` counter is reset to 2 whenever the player lands on the ground, which is detected within `check_collisions_y` when `landed_this_frame` becomes `true`.

Sound: A jump sound effect (`sfx_jump`) is played if a jump action is successfully performed.

3.3 Wall Mechanics

Wall sliding and wall jumping add significant depth to the player's movement capabilities.

Wall Slide Detection (`Player.update`):

Conditions: The check occurs only if the player is not on the ground (not `self.on_ground`) and is currently falling (`self.vel.y > 0`).

Collision Check: The player's hitbox (`self.rect`) is temporarily shifted one (or more, adjusted for power-up speed via `check_dist`) pixel horizontally towards the direction the player is pressing (left or right). `pygame.sprite.spritecollide` is used to check for collisions with platforms in this shifted position.

Activation: If a collision is detected and the player is holding the directional key into that wall, `self.wall_sliding` is set to `True`, and `self.wall_slide_side` is set to -1 (left) or 1 (right).

Wall Slide Effect:

If `self.wall_sliding` is `True`, the player's downward vertical velocity (`self.vel.y`) is capped at `PLAYER_WALL_SLIDE_SPEED`, slowing their descent.

The `jumps_left` counter is typically refreshed to 1, allowing one jump off the wall.

Wall Jump (Player.jump):

Activation: If `jump()` is called while `self.wall_sliding` is `True`.

Action: `self.wall_sliding` is set to `False`. A specific upward vertical velocity (`PLAYER_WALL_JUMP_Y_POWER`, modified by `jump_mult`) and a horizontal velocity away from the wall (`PLAYER_WALL_JUMP_X_POWER * -self.wall_slide_side`) are applied. `jumps_left` is set (typically to 1, as the wall jump consumes the initial charge).

3.4 Collision Detection

Accurate collision detection and response are vital for platformer gameplay. "The way of the shadow" uses Pygame's built-in sprite collision functions.

Method: `pygame.sprite.spritecollide(sprite, group, dokill)` is the primary function used. It checks for rectangle overlap between a single sprite (e.g., the player) and all sprites within a group (e.g., platforms, collectibles).

Player-Platform Collision:

Checked separately for X and Y axes in `Player.update` after applying velocity to each axis's position.

`check_collisions_x`: Detects horizontal overlap. If found, snaps the player's horizontal rect edge to the platform's edge and zeroes `self.vel.x`.

`check_collisions_y`: Detects vertical overlap. If the player is moving down and collides, snaps `rect.bottom` to `platform.rect.top`, zeroes `self.vel.y`, sets `on_ground = True`, and resets `jumps_left`. If moving up and hitting a ceiling, snaps `rect.top` to `platform.rect.bottom` and zeroes `self.vel.y`. Specific checks are included to ensure the player is appropriately aligned vertically with the platform edge to prevent side collisions being misinterpreted.

Player-Collectible Collision (Game.update):

`pygame.sprite.spritecollide(self.player, self.collectibles, True)` is used. The `dokill` argument is set to `True`, automatically removing the collected item from all sprite groups it belongs to.

The score is incremented, a sound is played, and the power-up logic is triggered.

Player-Goal Collision (`Game.update`):

`pygame.sprite.spritecollide(self.player, self.goal_group, False)` is used. `dokill` is `False` as the goal should remain.

Triggers the state transition to `STATE_LEVEL_COMPLETE` or `STATE_GAME_WON`.

4. Feature Implementation

Beyond the core mechanics, "The way of the Shadow" incorporates several features to create a more complete game experience. This section details their implementation.



4.1 Level System

The game supports multiple levels defined externally, allowing for easy expansion.

Data Structure (`levels.py`): A Python list or dictionary named `LEVELS` stores the data for each level. Each level entry is typically a dictionary containing keys like:

'platforms': A list of tuples, where each tuple defines a platform's (x, y, width, height).

'collectibles': A list of tuples defining each collectible's (x, y) center coordinates.

'goal': A tuple defining the goal's (x, y, width, height).

'player_start': A tuple defining the player's initial (x, y) coordinates for that level.

Loading (`Game.load_level(level_index)`):

Takes the index of the desired level.

Clears existing sprite groups (all_sprites, platforms, collectibles, goal_group).

Retrieves the corresponding level data from the LEVELS structure.

Iterates through the platform data, creating Platform instances and adding them to the relevant groups.

Iterates through collectible data, creating Collectible instances.

Creates the Goal instance.

Calls `self.player.reset()` with the level's player start coordinates.

Resets the level score (`self.score = 0`). Level timer (`level_elapsed_time`) is reset elsewhere (e.g., when starting the level or restarting).

4.2 Scoring and Collectibles

Collectibles provide an objective within each level and contribute to the power-up system.

Collectible Class (sprites.py): The Collectible sprite manages its own appearance and animation

Collection (Game.update): As described in Section 3.4, `pygame.sprite.spritecollide` detects player-collectible collisions.

Scoring: Upon collection, the `self.score` variable in the Game class is incremented. This score is primarily displayed on the HUD for the current level and is also used to track progress towards the power-up. It does not currently contribute to the high score (which is time-based).

Sound: A collection sound effect (`sfx_collect`) is played.

4.3 Timer System

Accurate timing is crucial for the speed-running aspect and high score feature.

Delta Time: The core game loop calculates `self.dt = self.clock.tick(FPS) / 1000.0`, representing the time elapsed (in seconds) since the last frame. This ensures timekeeping is relatively independent of fluctuations in frame rate.

Level Timer (`Game.level_elapsed_time`):

This float variable accumulates `self.dt` during the `STATE_PLAYING` state (if `self.timer_active`: check in `Game.update`).

It is reset to 0.0 when a level starts (`Game.events` when transitioning to `STATE_PLAYING` from menu/level complete) or when the level is restarted (`Game.events` handling 'R' key or restart button).

Its value upon reaching the goal is recorded for display and potential use.

Total Game Timer (`Game.total_game_time`):

This float variable accumulates the `level_elapsed_time` from successfully completed levels.

It is incremented in `Game.events` when the player clicks "Next Level" on the `STATE_LEVEL_COMPLETE` screen (`self.total_game_time += self.level_elapsed_time`).

It is reset to 0.0 only when starting a completely new game from the main menu (`Game.setup_game_variables`).

Final Time (`Game.final_time`):

Set when the player reaches the goal of the final level. Calculated as `self.total_game_time + current_level_final_time`.

Used for comparison against the high score.

Timer Activation (`Game.timer_active`): A boolean flag controlling whether `level_elapsed_time` increments. It's set to `True` when entering `STATE_PLAYING` and `False` upon reaching the goal, falling out, or pausing (if implemented).

Formatting (`format_time()`): A utility function converts total seconds (float) into a string formatted as `MM:SS:ms` for display on the HUD and end screens. Handles invalid values like `None` or `inf`.

4.4 High Score System

A persistent high score encourages replayability.

Storage: The best completion time (lowest `final_time`) is stored as a floating-point number (representing seconds) in a simple text file (`highscore.txt`, path defined in `settings.py`). `float('inf')` is used to represent no score set yet.

Loading (`Game.load_highscore()`):

Called during `Game.__init__` and potentially when returning to the menu.

Attempts to read the value from `highscore.txt`.

Includes error handling for file not found (creates the file with `'inf'`), invalid content (non-numeric, zero, or negative values are treated as `'inf'`), or empty file.

Returns the loaded score (or `float('inf')`).

Saving (`Game.save_highscore()`):

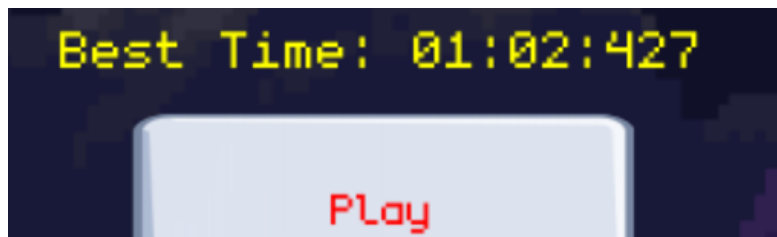
Called only when the player wins the game (`STATE_GAME_WON`) and their `final_time` is less than the current `self.high_score`.

Writes the new best time (as a string) to `highscore.txt`, overwriting the previous content.

Includes basic error handling for file writing issues.

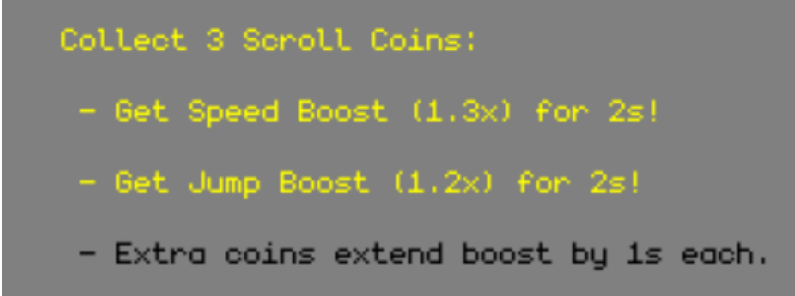
Comparison: In the `STATE_GAME_WON` update logic, `self.final_time` is compared to `self.high_score`. If the new time is better, `self.high_score` is updated in memory before calling `save_highscore()`.

Display: The formatted best time is displayed on the main menu (`STATE_MENU`) and on the game won screen (`STATE_GAME_WON`) for comparison.



4.5 Power-Up System

The power-up adds a dynamic element triggered by collecting scrolls.



```
Collect 3 Scroll Coins:  
- Get Speed Boost (1.3x) for 2s!  
- Get Jump Boost (1.2x) for 2s!  
- Extra coins extend boost by 1s each.
```

Tracking (Game class):

coins_for_powerup_count: Integer tracking scrolls collected towards the next power-up activation. Reset on activation or level restart after death.

powerup_active: Boolean flag indicating if the boost is currently active.

powerup_end_time: Stores the `pygame.time.get_ticks()` timestamp when the active power-up should expire.

Activation (Game.update within collectible collision):

When `coins_for_powerup_count` reaches `COINS_NEEDED_FOR_POWERUP` (from `settings.py`):

`powerup_active` is set to `True`.

`powerup_end_time` is calculated as `current_time + POWERUP_INITIAL_DURATION`.

`coins_for_powerup_count` is reset to 0.

Extension (Game.update within collectible collision):

If a coin is collected while `powerup_active` is already `True`:

`powerup_end_time` is extended by `POWERUP_EXTENSION_PER_COIN`.

Expiration (Game.update):

Checks if `powerup_active` is `True` and if `pygame.time.get_ticks()` is greater than or equal to `powerup_end_time`.

If expired, `powerup_active` is set back to `False`.

Effect Application (Player.update, Player.jump):

Multiplier variables (`speed_mult`, `jump_mult`) are determined based on `self.game.powerup_active`.

These multipliers are applied to `PLAYER_ACC`, `MAX_RUN_SPEED`, `PLAYER_JUMP_POWER`, `PLAYER_DOUBLE_JUMP_POWER`, and `PLAYER_WALL_JUMP_Y_POWER` when calculating physics. The wall slide check distance (`check_dist`) was also adjusted based on the power-up state to ensure reliable collision detection at higher speeds.

Reset: The power-up state (`coins_for_powerup_count`, `powerup_active`, `powerup_end_time`) is reset in `Game.setup_game_variables` (when starting a new game from menu) and specifically within `Game.events` when restarting a level after death or via the 'R' key during play.

UI Indication (`Game.draw_playing`): Text is displayed on the HUD showing remaining boost time when active, or progress towards the next boost otherwise.

4.6 User Interface (UI)

The UI provides necessary information and interaction points for the player.

HUD (`Game.draw_playing`): Displays level number, scrolls collected (`self.score`), the current running timer, and power-up status/progress. Uses the `draw_text` utility function.

Menus & End Screens (`Game draw helpers`): Specific methods (`draw_menu`, `draw_controls`, `draw_level_complete`, etc.) handle rendering text and buttons appropriate for each game state. They often use `draw_text` and the `Button` class. An overlay effect (`draw_end_screen_overlay`) darkens the background on end screens.

Text Rendering (`ui.py: draw_text`): A utility function simplifies drawing text to the screen, handling font rendering, positioning (including centering), and color.

Buttons (`ui.py: Button`): The `Button` class encapsulates button logic:

Stores position, text, font, and images (normal/hover).

draw() method renders the button image and text.

check_hover() updates the button's appearance based on mouse position.

is_clicked() checks Pygame events to determine if the button was clicked by the mouse.

4.7 Animation System

Simple sprite animation adds life to the player and collectibles.

Spritesheets: Animations are derived from spritesheet images where multiple frames are laid out in a single file. Paths are defined in settings.py.

Frame Extraction (Player._extract_frames): This helper method loads a spritesheet, calculates individual frame dimensions, iterates through the sheet, uses `spritesheet.subsurface()` to extract each frame, scales it to the desired visual size (`PLAYER_WIDTH/HEIGHT`), and appends it (and its horizontally flipped version) to the appropriate animation list (e.g., `idle_frames_r`, `run_frames_l`). Includes error handling and fallback frame generation.

Player Animation (Player.animate):



Determines the `current_action` ('idle' or 'run') based on horizontal velocity.

Selects the correct frame list based on `current_action` and `facing_right`.

Uses `pygame.time.get_ticks()` and `last_anim_update` along with `PLAYER_ANIMATION_SPEED` to control frame timing.

Cycles through `current_frame_index` using the modulo operator (%).

Updates `self.image` to the new frame. Resets `current_frame_index` if the action changes.

Collectible Animation (Collectible.update): Similar timing logic using `last_anim_update` and `COLLECTIBLE_ANIM_SPEED` to cycle through its frames list and update `self.image`.

4.8 Audio System

Sound enhances the game's atmosphere and provides feedback.

Initialization: `pygame.mixer.pre_init()` and `pygame.mixer.init()` are called early in `Game.__init__` for optimal sound setup.

Background Music:

Loaded using `pygame.mixer.music.load()` in `Game.load_assets`.

Played on a loop using `pygame.mixer.music.play(loops=-1)` in `Game.run`.

Volume set using `pygame.mixer.music.set_volume()`.

Stopped using `pygame.mixer.music.stop()` when the game exits. Rewound using `pygame.mixer.music.rewind()` when returning to the menu.

Sound Effects:

Loaded as `pygame.mixer.Sound` objects (e.g., `self.sfx_jump`, `self.sfx_collect`) in `Game.load_assets`.

Volume set individually using `sound.set_volume()`.

Played using `sound.play()` via the `Game.play_sound()` helper method, which includes basic error handling. Triggered by specific game events (jumping, collecting).

5. Asset Management

This section covers the types of assets used in "The way of the shadow" and how they are managed within the project structure.

5.1 Graphics

Visual elements are crucial for the game's appearance.

Player: Animated spritesheets (player_idle.png, player_run.png) containing multiple frames for different actions. Extracted and scaled in Player.load_images.

Platforms: A single tile image (platform_tile.png) is loaded and tiled across the dimensions of Platform objects in Platform.__init__.

Collectibles: An animated spritesheet or sequence of images (scroll_0.png, scroll_1.png, etc.) loaded into a list of frames for the Collectible class.

Goal: A static image (door.png) representing the level exit, scaled to fit the Goal object's dimensions.

Background: A static background image (background.png) loaded and displayed covering the entire screen.

UI Elements: Images for buttons (button_normal.png, button_hover.png) used by the Button class.

Loading: All images are loaded using pygame.image.load() within the Game.load_assets method (or methods called by it, like Player.load_images). Paths are constructed using os.path.join() and filenames/directories from settings.py. .convert() or .convert_alpha() is used for performance optimization and transparency handling. Scaling (pygame.transform.smoothscale) is applied where necessary to match defined dimensions (e.g., PLAYER_WIDTH, BUTTON_DISPLAY_WIDTH). Fallback surfaces (e.g., colored rectangles) are created if image loading fails.

5.2 Audio

Audio assets provide auditory feedback and atmosphere.

Background Music: A single music track (`music_background.ogg` or `.mp3`) loaded via `pygame.mixer.music`.

Sound Effects: Short audio files (`sfx_jump.wav`, `sfx_collect.wav`) loaded as `pygame.mixer.Sound` objects. `.wav` format is generally recommended for short effects due to lower latency, while `.ogg` or `.mp3` are suitable for longer music tracks due to compression.

Loading: Audio files are loaded in `Game.load_assets`, with paths constructed similarly to images using `settings.py`. Error handling is included to prevent crashes if files are missing.

5.3 Configuration (`settings.py`)

The `settings.py` file acts as a central configuration hub, decoupling constants and tuning parameters from the main game logic. This includes:

Screen dimensions, FPS, basic colors.

File paths for all assets (images, sounds, fonts, high score file).

Player physics parameters (gravity, acceleration, friction, speeds, jump powers, wall slide speed).

Player visual and hitbox dimensions.

Animation speeds for player and collectibles.

Game state constants.

Power-up parameters (coins needed, duration, extension, multipliers).

UI element properties (font sizes, button dimensions).

Using this file makes it significantly easier to adjust game balance, change assets, or modify core parameters without digging through multiple logic files.

6. Testing and Debugging

Ensuring the game functions correctly and identifying/fixing bugs were critical parts of the development process.

```
Wall slide check. SpeedMult: 1, CheckDist: 1
  Checking right wall (1px)... Hits_R: False
  Checking left wall (1px)... Hits_L: False
Wall slide check. SpeedMult: 1, CheckDist: 1
  Checking right wall (1px)... Hits_R: False
  Checking left wall (1px)... Hits_L: False
GOAL HIT: Pausing timer.
Level 1 finished. Time for level: 00:05:915. State: 3
Power-up Expired.
NEXT LEVEL: Added 5.915s. New total = 5.915s
Player reset. Hitbox: (50, 615), OnGround: False
Level 2 loaded. Total time before this level: 5.915s
Level 2 loaded. Coins towards powerup: 0, Active: False
Wall slide check. SpeedMult: 1, CheckDist: 1
  Checking right wall (1px)... Hits_R: False
```

6.1 Testing Approach

Testing was primarily conducted through manual playtesting due to the dynamic nature of gameplay interactions. The approach involved:

Incremental Testing: Testing features as they were implemented (e.g., testing basic movement before adding jumping, testing jumping before wall sliding).

Core Mechanic Testing: Repeatedly performing basic actions (running, jumping, double jumping) in various scenarios (on flat ground, near edges, different speeds) to ensure responsiveness and consistency.

Advanced Mechanic Testing: Specifically designing test cases for wall sliding and wall jumping on different wall configurations and approach angles. Testing jump heights and distances.

Level Playthroughs: Playing through each level from start to finish to check for playability, unreachable areas, collision issues, and goal triggering.

Feature Verification: Systematically checking specific features:

Collectibles: Ensuring they disappear on contact and update the score/power-up count.

Timer: Checking accuracy against external timer, correct display formatting, proper pausing/resetting.

High Score: Verifying saving of new best times and correct loading/display on menu/win screen. Testing edge cases like no score file or invalid score file.

Power-Up: Testing activation at the correct count, duration timing, extension mechanic, visual/audio feedback, correct application of physics multipliers, and proper reset on death/restart.

State Transitions: Ensuring smooth and correct transitions between menu, playing, game over, level complete, controls, and win states via buttons and game events.

UI Interaction: Testing button hover effects and click registration.

Edge Case Testing: Attempting unusual actions or sequences (e.g., jumping immediately after landing, hitting corners, trying to wall slide on very small surfaces) to uncover potential bugs.

6.2 Debugging Techniques

Several techniques were employed to identify and resolve issues:

`print()` Statements: Extensive use of `print()` statements within update methods (`Player.update`, `Game.update`) to output the values of key variables in real-time to the console (e.g., player position, velocity, acceleration, `on_ground` status, `wall_sliding` state, `jumps_left`, timer values, power-up state variables, collision check results). This was invaluable for understanding state changes frame-by-frame.

Visual Debugging (Hitbox Drawing): Temporarily drawing the outlines of sprite rectangles (`self.rect`) directly onto the screen using `pygame.draw.rect(screen, color, rect, width)`. This allowed visual confirmation of collision boundaries for the player, platforms, and other objects, helping to diagnose collision detection failures or unexpected interactions. This was particularly useful for the wall slide implementation check.

Code Isolation: Temporarily commenting out sections of code (e.g., power-up physics modifications) to isolate whether a specific feature was causing an unrelated bug (as was done temporarily to diagnose the wall slide issue after power-ups were added).

Error Message Analysis: Carefully reading traceback messages generated by Python upon crashes to identify the location and type of error (e.g., FileNotFoundError, AttributeError, ValueError).

Logical Stepping: Mentally tracing the code execution flow for specific scenarios (e.g., what happens frame-by-frame during a wall jump) to identify potential logical flaws.

6.3 Key Challenges and Solutions

Several challenges were encountered during development:

Wall Slide/Jump Reliability:

Challenge: Initially, wall slide detection was inconsistent, sometimes failing to trigger or dropping unexpectedly. Wall jumps felt unresponsive.

Solution: Refined the detection logic in `Player.update` by checking conditions (not on_ground, `vel.y > 0`, key press into wall) rigorously. Implemented the temporary 1-pixel (later adjusted `check_dist`) horizontal shift for collision checking against the wall, which proved crucial. Ensured `jumps_left` was correctly refreshed during the slide and managed during the wall jump. Debugging with print statements and hitbox visualization was essential. The issue reappeared slightly after adding the speed boost power-up, requiring an increase in the collision check distance (`check_dist`) to account for higher velocities potentially skipping the single-pixel check.

Collision Response Precision (Getting Stuck/Passing Through):

Challenge: Occasionally, the player might slightly sink into the ground or get stuck on edges. Early versions might have had issues with passing through thin platforms at high speeds.

Solution: Implemented separate X and Y axis collision checks and responses. Snapping the player's rect edge directly to the platform edge upon collision prevents sinking. Ensuring collision checks happen after position updates for each axis helps resolve issues. While not fully implemented here, more advanced techniques like continuous collision detection could be considered for extremely high speeds. Added vertical alignment checks in `check_collisions_y` to prevent side collisions triggering ground landings.

Power-Up State Management:

Challenge: Ensuring the power-up counter (coins_for_powerup_count) reset correctly upon death/restart, rather than persisting incorrectly. Making sure the timer activated, extended, and deactivated accurately.

Solution: Added explicit reset logic for power-up variables (coins_for_powerup_count, powerup_active, powerup_end_time) in the relevant event handling blocks within Game.events (specifically for STATE_GAME_OVER restart and STATE_PLAYING 'R' key press). Used pygame.time.get_ticks() consistently for timing comparisons.

Asset Loading Errors:

Challenge: Game crashing if image or sound files were missing or corrupted.

Solution: Wrapped asset loading calls (pygame.image.load, pygame.mixer.Sound, etc.) in try...except blocks within Game.load_assets and helper methods like Player._extract_frames. Implemented fallback mechanisms (e.g., creating solid color surfaces, using dummy sound objects, printing warning messages) to allow the game to continue running, albeit with missing assets, aiding debugging.

7. Results and Discussion

This section evaluates the success of the "The way of the Shadow" project in meeting its objectives and discusses the overall outcome and gameplay experience.

7.1 Achieved Objectives

The project successfully met the majority of the goals outlined in the introduction:

Core & Advanced Mechanics: Running, jumping, double jumping, wall sliding, and wall jumping were implemented and function as intended, providing a solid platforming base. Physics feel reasonably responsive after tuning.

Level System: A functional system for loading multiple levels from levels.py was created.

Collectibles & Scoring: Scroll Coins can be collected, incrementing a visible score counter.

Game State Management: A state machine effectively manages transitions between menu, gameplay, controls, and various end screens.

Timer & High Score: Both level and total game timers function correctly, using delta time for accuracy. The high score system successfully loads, saves, and compares times, persisting data between sessions.

Power-Up System: The speed/jump boost power-up activates upon collecting the required scrolls, provides the intended physics modifications for a set duration, extends correctly with further collection, and resets appropriately on death/restart.

User Interface (UI): A functional menu, controls screen, HUD, and end screens were implemented with working buttons and text displays.

Audiovisuals: Basic background music, sound effects for key actions, and player idle/run animations were successfully integrated.

Modularity & Readability: The codebase is organized into distinct modules (main.py, sprites.py, settings.py, levels.py, ui.py), enhancing readability and maintainability. OOP principles were used for game objects and UI elements.

7.2 Gameplay Experience

"The way of the Shadow" offers a challenging and potentially fast-paced platforming experience.

Strengths:

The combination of double jumping and wall mechanics allows for expressive and fluid movement, enabling skilled players to navigate levels quickly.

The timer and high score system provide a clear objective for speed-running and replayability.

The power-up system adds a dynamic element, rewarding players for thorough collection with temporary bursts of speed and agility, changing how sections might be approached.

The controls are generally responsive.

Areas for Improvement:

Level Design: The included levels are basic proofs of concept. More intricate level design utilizing the advanced mechanics more deliberately would significantly enhance engagement. Difficulty scaling across levels could be improved.

Visual Feedback: Lack of specific animations for jumping, falling, and wall sliding means player actions are sometimes conveyed only by movement changes, not visual cues. More animations would improve clarity and feel.

Game Feel: Further tuning of physics parameters (gravity, jump heights, friction, power-up multipliers) could refine the "feel" of the controls. Adding subtle effects like particle trails, screen shake (on hard landings), or "coyote time" (allowing a jump shortly after running off a ledge) could enhance game feel.

Lack of Enemies/Hazards: The absence of threats reduces the challenge in some aspects, making it primarily a navigation and time-trial game.

7.3 Code Quality and Maintainability

The project structure and coding practices contribute positively to maintainability:

Modularity: Separating logic into different files (main, sprites, settings, levels, ui) makes it easy to locate and modify specific parts of the game.

Configuration: Using settings.py for constants allows for easy tuning of game parameters without altering core logic code.

OOP: Defining classes for game entities (Player, Platform, etc.) encapsulates data and behavior logically.

Readability: Code includes comments explaining key sections and logic. Variable and function names are generally descriptive.

Potential Improvements:

Further refactoring could potentially reduce coupling between the Game class and the Player class (e.g., passing specific game state information to Player.update instead of the entire game object, though the current approach is common and practical in Pygame).

More comprehensive error handling could be added in some areas.

Implementing unit tests (though challenging for highly graphical/interactive components) could improve robustness for specific logic modules if the project grew larger.

Overall, the codebase provides a solid foundation for future development and maintenance.

8. Future Work and Improvements

While "The way of the shadow" meets its primary objectives as a prototype, there are numerous avenues for future development and enhancement.

8.1 Potential Enhancements

Enemies and Hazards: Introduce enemy sprites with basic AI (e.g., patrolling, chasing) and environmental hazards (spikes, moving platforms, lasers) to increase challenge and engagement. This would require implementing health/damage systems and potentially combat mechanics.

More Levels & Themes: Develop a larger set of levels with increasing complexity and potentially different visual themes (e.g., caves, forests, castles). Introduce new platform types or level-specific mechanics.

Expanded Power-Ups: Design and implement different types of power-ups (e.g., temporary invincibility, projectile attack, altered gravity) beyond the current speed/jump boost.

Enhanced Animations & Visuals: Create and integrate more detailed sprite animations for the player (jump, fall, wall slide, attack) and potentially for enemies. Improve background art and add particle effects (dust on landing, speed lines, coin collection sparkle).

Improved Sound Design: Add more sound effects for various interactions (landing, wall sliding, hitting ceiling, power-up activation/expiration) and potentially different music tracks for different levels or game states.

Level Editor: Develop a simple tool to allow users (or the developer) to create and save custom levels more easily than manually editing the levels.py file.

Pause Menu: Implement a pause state (`STATE_PAUSED`) accessible during gameplay, allowing players to suspend the action and potentially access options or return to the menu.

Controller Support: Add input handling for game controllers using Pygame's joystick module.

Story/Narrative: Integrate a simple storyline or context for the player's actions through intro/outro screens or in-game text elements.

8.2 Refactoring Opportunities

State Management: For a larger game with more states, consider implementing a more formal State pattern using dedicated state classes instead of if/elif blocks based on constants in the main loop.

Physics Engine: While Pygame's vectors and basic physics simulation work well here, for more complex physics interactions, integrating a dedicated 2D physics engine library (like Pymunk) could be beneficial, though it would add complexity.

Event Handling: Centralize event handling further, perhaps using an event bus or dispatcher pattern if interactions become very complex.

Hardcoded Values: Ensure all tunable parameters are moved to settings.py and that no "magic numbers" remain directly in the logic code.

9. Conclusion / Final Thoughts

9.1 Summary of Project and Achievements

The "The way of the Shadow" project successfully resulted in a functional and engaging 2D platformer prototype built with Python and Pygame. It demonstrates the implementation of essential platforming mechanics, including advanced techniques like wall sliding and wall jumping, alongside key game features such as a multi-level structure, collectibles, a time-based high score system, and a dynamic power-up mechanic. The project adhered to a modular design, promoting code organization and maintainability. The objectives set at the outset were largely achieved, providing a solid foundation that showcases core game development principles.

9.2 Learning Outcomes

This project provided significant learning experiences in several areas:

Pygame Proficiency: Gained practical skills in using various Pygame modules for rendering, sprite handling, event management, collision detection, audio, and timing.

Game Loop Architecture: Developed a deeper understanding of the fundamental event-update-draw game loop structure and state management techniques.

2D Physics Simulation: Learned to implement basic physics concepts like gravity, acceleration, friction, and velocity using vectors for platformer movement.

Collision Detection & Response: Acquired experience in implementing robust collision detection for different axes and handling the resulting interactions.

Object-Oriented Game Design: Applied OOP principles to structure game entities and systems effectively.

Feature Integration: Gained experience integrating various distinct features (timer, high score, power-ups) into a cohesive whole.

Debugging Techniques: Enhanced debugging skills through systematic use of print statements, visual debugging, and logical problem-solving, particularly when tackling subtle physics or state-related bugs like the wall sliding issue.

Asset Pipeline: Understood the process of creating/acquiring assets (graphics, audio) and integrating them into the game engine.

9.3 Final Remarks

"The way of the shadow" serves as a strong demonstration of the capabilities of Python and Pygame for creating interactive 2D games. While there is ample scope for expansion and polish, the current implementation provides a complete core gameplay loop with several engaging features. The development process was iterative, involving problem-solving and refinement, particularly around the physics and state interactions. The final prototype is playable, achieves its primary goals, and represents a valuable learning experience in game development.

Bibliography / References

Python Software Foundation. (Ongoing). Python Language Reference (Version 3.x). Retrieved [Date You Accessed Python Docs, e.g., October 26, 2023], from <https://www.python.org>

Pygame Developers. (Ongoing). Pygame Documentation (Version 2.x). Retrieved [Date You Accessed Pygame Docs, e.g., October 26, 2023], from <https://www.pygame.org/docs/>

Game assets :

All the icons/sprites -

Top game assets. (n.d.). itch.io. <https://itch.io/game-assets>

Game audio -

Top game assets tagged Audio. (n.d.). itch.io. <https://itch.io/game-assets/tag-audio>

For Fonts:

monogram by datagoblin. (n.d.). itch.io. <https://datagoblin.itch.io/monogram>

Appendices

Appendix A: Key Code Snippets

Player Physics Update (Snippet from Player.update)

```
# Inside Player.update() after input/acceleration/friction...
    # Update Velocity
    self.vel += self.acc
```

```
    # Cap Horizontal Speed
    current_max_speed = MAX_RUN_SPEED * speed_mult
    if abs(self.vel.x) > current_max_speed:
self.vel.x = current_max_speed if self.vel.x > 0 else -current_max_speed
    if abs(self.vel.x) < 0.1: self.vel.x = 0
```

```
    # Horizontal Movement & Collision
    self.pos.x += self.vel.x + 0.5 * self.acc.x
    self.rect.x = round(self.pos.x)
    self.check_collisions_x(platforms)
```

```
    # Vertical Movement & Collision
    self.pos.y += self.vel.y + 0.5 * self.acc.y
    self.rect.y = round(self.pos.y)
    if not self.wall_sliding: self.on_ground = False
```

```
self.check_collisions_y(platforms)
```

```
# Apply Max Fall Speed AFTER Y collisions
```

```
if self.vel.y > MAX_FALL_SPEED: self.vel.y = MAX_FALL_SPEED
```

```
Use code with caution.
```

```
Python
```

Example: Wall Slide Detection Logic (Snippet from Player.update)

```
# Inside Player.update()
```

```
# ...
```

```
check_dist = 2 if original_speed_mult > 1.0 else 1 # Adjusted check distance
```

```
if not self.on_ground and self.vel.y > 0:
```

```
self.rect.x += check_dist; hits_r = pygame.sprite.spritecollide(self, platforms, False);
```

```
self.rect.x -= check_dist
```

```
if hits_r and (keys[pygame.K_RIGHT] or keys[pygame.K_d]):
```

```
self.wall_sliding = True; self.wall_slide_side = 1
```

```
elif (keys[pygame.K_LEFT] or keys[pygame.K_a]):
```

```
self.rect.x -= check_dist; hits_l = pygame.sprite.spritecollide(self, platforms, False);
```

```
self.rect.x += check_dist
```

```
if hits_l:
```

```
self.wall_sliding = True; self.wall_slide_side = -1
```

```
if self.wall_sliding:
```

```
self.vel.y = min(self.vel.y, PLAYER_WALL_SLIDE_SPEED);
```

```
self.jumps_left = 1
```

Power-up Activation Logic (Snippet from Game.update)

```
# Inside Game.update() collectible collision block...
```

```
# ... (loop through collected items) ...
```

```
if self.powerup_active:
```

```
self.powerup_end_time += POWERUP_EXTENSION_PER_COIN
```

```
# ... (print message)
```

```
else:
```

```
self.coins_for_powerup_count += 1
```

```
# ... (print message)
```

```
if self.coins_for_powerup_count >= COINS_NEEDED_FOR_POWERUP:
```

```
self.powerup_active = True
```

```
self.powerup_end_time = current_time + POWERUP_INITIAL_DURATION
```

```
self.coins_for_powerup_count = 0
```

```
# ... (print message)
```

Appendix B: Level Data Example (levels.py)

```
# levels.py excerpt (Level 1 Data)
LEVEL_1 = {
    'platforms': [
        (0, SCREEN_HEIGHT - 40, SCREEN_WIDTH, 40),      # Ground
        (200, SCREEN_HEIGHT - 180, 150, 20),            # Mid platform 1
        (SCREEN_WIDTH - 300, SCREEN_HEIGHT - 300, 100, 20), # Mid platform 2
        (500, SCREEN_HEIGHT - 450, 200, 20),            # High platform
        (0, 200, 50, SCREEN_HEIGHT - 240),              # Left Wall
        (SCREEN_WIDTH - 50, 100, 50, SCREEN_HEIGHT - 200), # Right Wall
    ],
    'collectibles': [
        (250, SCREEN_HEIGHT - 220), # On mid platform 1
        (SCREEN_WIDTH - 250, SCREEN_HEIGHT - 340), # On mid platform 2
        (600, SCREEN_HEIGHT - 490), # On high platform
        (100, SCREEN_HEIGHT - 80), # Near start on ground
        (SCREEN_WIDTH - 100, SCREEN_HEIGHT - 80), # Near end on ground
    ],
    'goal': (SCREEN_WIDTH - 100, 60, 50, 40), # Top right near wall
    'player_start': (100, SCREEN_HEIGHT - 100) # Start left on ground
}

LEVELS = [
    LEVEL_1,
    # Add LEVEL_2, LEVEL_3 etc. dictionaries here
]

MAX_LEVELS = len(LEVELS)
```

