uc3m | Universidad Carlos III de Madrid

DB in Telecomunication Technologies and Data Science

Programming 24/25

Group 196

Final Project
"Mario Bros."

Samuel Matamoros Alonso — 100583032

Darío Castro Vila — 100581710

Professor

Ángel García Olaya

Table of contents

1. Summary

This document describes the design and implementation of a 2D game inspired by Nintendo's Mario Bros. LCD Game & Watch title, developed in Python using the Pyxel retro game engine.

The report explains the main classes, core algorithms (movement, collision, state management), the development process and decisions taken, as well as the current functionality, missing features, and extra improvements added over the basic requirements.

2. Introduction

The goal of this project was to design and implement a small but complete video game in Python, applying object-oriented programming, basic game loop structure and state management. The chosen concept is a recreation of the classic LCD Mario Bros. factory game: Mario and Luigi move boxes on conveyor belts to load a truck, while avoiding drops and handling increasing difficulty.

Pyxel was selected as the main library because it provides a simple framework for pixel-art games: screen management, sprites, keyboard input and a fixed-rate update/draw loop. The project also aims to practice modular design by separating the code into classes such as Board, Character, Conveyor, Package and Truck, plus a configuration module config for constants and sprites.

2.1. Game description

The game takes place in a factory where Mario and Luigi must move boxes along several conveyor belts to load them into a truck. The player controls each character's vertical position so they can catch and pass boxes in time, avoiding breaks; each delivered box increases the score, while each broken box increases the fail counter, and the game ends when three fails are reached.

3. Design and Architecture

3.1. Main

Application entry point and integration with the Pyxel engine.

- Attributes:
  - ‣ board: Board - Main game controller.

- Most relevant methods:
  - ‣ init() - Initializes Pyxel (window, FPS, resources) and creates the Board instance.
  - ‣ update() – Called every frame by Pyxel; forwards to board.menu_update() and board.update() when the menu is not active and the game is not over.
  - ‣ draw() – Clears the screen, calls board.draw(), and, if needed, draws the menu or the game-over image.

3.2. Board

Central game controller. Manages state, difficulty, characters, conveyors, packages, truck, boss and menu.

- Main Attributes:
  - ‣ Game state: difficulty, score, fails, game_over.
  - ‣ UI state: menu_active, menu_selected.
  - ‣ Entities: mario: Character, luigi: Character, truck: Truck, conveyors: list[Conveyor], packages: list[Package].
  - ‣ Difficulty / progression: number_of_conveyors, conveyor_speed, number_of_packages, points_for_package, number_of_deliveries.
  - ‣ Boss system: boss_active, boss_target, boss_timer.
- Most relevant methods:
  - ‣ Boss system: boss_active, boss_target, boss_timer.
  - ‣ menu_draw() / top_menu() – Draw menu and HUD (score, fails, buttons).
  - ‣ difficulty0(), difficulty1(), difficulty2(), difficulty3() – Configure conveyors, speeds and scoring per difficulty
  - ‣ update() – Per-frame logic: check difficulty, update truck delivery and boss punishment, and when no special state is active update characters, generate and move packages, and check game-over.
  - ‣ draw() – Draw conveyors, packages, platforms, truck, characters, boss, and UI.

3.3. Character

Represents Mario or Luigi, with movement and sprite/animation.

- Main Attributes:
  - ‣ character: str – "MARIO" or "LUIGI".
  - ‣ level: int – Current vertical level/platform.
  - ‣ has_package: bool – True if carrying a box.
  - ‣ resting: bool – True during truck deliveries.

‣ reprimand: bool – True when being punished by the boss.
- Most relevant methods:
    ‣ init(character: str) – Initialize character type and default state.
    ‣ update(max_level: int) – Handle input (different keys for Mario and Luigi) and change level when not resting or reprimanded; update animation state.
    ‣ draw() – Draw character sprite on screen according to level and current state.

3.4. Conveyor

Represents a conveyor belt with a given speed.

- Main Attributes:
    ‣ level: int – Conveyor index/height-
    ‣ speed: float – Movement speed for packages on this belt.
- Most relevant methods:
    ‣ init(level: int, speed: float) – Configure conveyor level and speed.
    ‣ draw(game_over: bool) – Draw conveyor tiles using config sprites.

3.5. Package

Represents a moving box on the conveyors.

- Main Attributes:
    ‣ level: int – Current conveyor level.
    ‣ speed: float – Horizontal movement speed.
    ‣ state: str – "CONVEYOR", "HANDLED", "BROKEN" or "TRUCK"
- Most relevant methods:
    ‣ update() – Move the package according to its speed and state.
    ‣ at_the_end() -> bool – Check if the package has reached the end of its current conveyor.
    ‣ move_to_next_conveyor() – Move package to the next conveyor when handled correctly.
    ‣ broken() – Mark the package as broken.
    ‣ draw() – Draw the package with the appropriate sprite.

3.6. Truck

Collects delivered packages and manages the delivery timing.

- Main Attributes:
    ‣ number_of_packages: int – Number of boxes loaded (0–8).
    ‣ state: str – "LOADING" or "DELIVERY".
    ‣ delivery_start_frame: int | None – Frame when delivery started (for timing).
- Most relevant methods:
    ‣ start_delivery() – Switch to "DELIVERY" and record current frame.
    ‣ delivery_done(duration_frames: int = 180) -> bool – Check if delivery time has elapsed.
    ‣ finish_delivery() – Return to "LOADING", reset timer and empty the truck.

▸ draw(level: int) – Draw truck head and bed depending on number_of_packages.

4. Main algorithms

4.1. Game loop

The game loop is managed by Pyxel, which repeatedly calls Main.update() and Main.draw() at a fixed frame rate (60 FPS). Main.update() delegates to Board, which first updates the menu, then calls Board.update() only when the menu is closed and the game is not over, ensuring a clean separation between menu and gameplay.

4.2. Character movement and input

Character movement is handled in Character.update(max_level), where keyboard input is read using pyxel.btnp.
- Mario uses up/down arrow keys to change level; Luigi uses W/S.
- Movement is allowed only if the character is not resting and not reprimanded, and the new level stays within 0 and max_level / 2 - 1.

This algorithm ensures simple, discrete vertical movement that matches the conveyor layout.

4.3. Package update and handling algorithm

The package update algorithm in Board.package_update_all() iterates over all active Package objects and performs three main steps:
- 1) Synchronize each package's speed with the Conveyor speed at the current level and call package.update().
- 2) If package.at_the_end() is true, check:
    - If it is at the final conveyor: Luigi can load it into the truck; otherwise it breaks.
    - If it is on an even conveyor: Mario must be at the matching level to pass it; otherwise it breaks.
    - If it is on an odd conveyor: Luigi must be at the matching level to pass it; otherwise it breaks.
- 3) When a box is handled correctly, increase score and move it to the next conveyor; when it breaks, increase fails and trigger the boss punishment for the responsible character.

Packages in "BROKEN" or "TRUCK" state are removed from the list to keep the system clean.

4.4. Truck delivery timing

When Truck.number_of_packages reaches a chosen threshold, Board.truck_delivery() is called:
- Score is increased, deliveries counter is updated and the truck is emptied.
- Truck.start_delivery() sets the state to "DELIVERY" and remembers the current frame.
- Characters are set to resting = True and their package flags are cleared.

In Board.update(), as long as the truck is delivering, the algorithm checks Truck.delivery_done(). Once the delivery time passes, Truck.finish_delivery() is called and the resting flags are cleared, returning to normal gameplay.

4.5. Boss punishment system

The boss system is a simple state machine controlled by boss_active, boss_target and boss_timer in Board:

- On a failure (broken box), if no punishment is currently active, boss_active is set to true, boss_target is set to "MARIO" or "LUIGI" depending on who failed, and boss_timer is initialized to a certain number of frames.
- While boss_active is true, boss_timer decreases each frame, the corresponding character's reprimand flag is set and the other one is cleared, and normal gameplay updates (movement and packages) are skipped.
- When boss_timer reaches zero, boss_active is set to false and both reprimand flags are cleared, allowing the game to continue.

This avoids nested or overlapping punishments and guarantees that every punishment eventually ends.

5. Work done and functionality

5.1. Implemented functionality

- Complete game loop and screen using Pyxel (initialization, update, draw)
- Two playable characters (Mario and Luigi) with separate controls and level-based movement.
- Multiple conveyors with configurable speeds per difficulty.
- Packages that move along conveyors, can be passed between characters and loaded into the truck, or break if missed.
- Scoring and fail system with game-over when reaching three fails.
- Difficulty selection menu that configures number of conveyors, speeds and scoring parameters.
- Truck that fills with boxes and triggers a timed delivery state during which characters rest.
- Boss punishment system: boss appears, the failing character is moved to a fixed position and shows reprimand sprites for a short time, pausing gameplay.
- Simple sound effects.

6. User manual

- Starting the game:
  - ‣ Launch the Python script that instantiates Main (e.g., python main.py).
  - ‣ The game window opens with the default difficulty and the possibility to open the menu.
- Controls:
  - ‣ General:
    - – Q: quit the game
    - – M: open/close the difficulty menu
    - – ENTER: confirm the selected difficulty in the menu.
  - ‣ Mario/Luigi
    - – Arrow UP / W: move character up.
    - – Arrow DOWN / S: move character down.

7. Conclusions

7.1. Final summary

The project successfully delivers a small but complete 2D game inspired by the Mario Bros. Game & Watch factory game, implemented in Python using Pyxel. It demonstrates object-oriented design with classes for the main game controller, characters, conveyors, packages and truck, as well as the use of a simple configuration module for sprites and constants.

7.2. Main problems encountered

Several issues related to game states were encountered, including freezes when the boss or the truck entered special modes and the main loop stopped updating the logic needed to exit those modes. Additional problems appeared with repeated boss activation during consecutive fails, incorrect character movement while being reprimanded, and overly restrictive package spawn conditions that could leave the game with no new boxes. The package logic was the hardest part to implement in the game, anyway implementing the individual features wasn't hard, combining them to work properly was the hard part of the project.

7.3. Personal evaluation and improvements

The project was a good exercise in structuring a game using classes and in handling state machines and timers in a real-time loop. If the project were extended, priority improvements would include adding richer animations for characters and boss, a persistent high-score system, and a more adaptive difficulty that reacts to the player's performance.

7.4. Issues and solutions

Several typical state-management issues appeared during development, such as the game "freezing" when the boss or truck entered a special state and the main loop stopped updating the logic that should exit that state. This was solved by centralizing state control inside Board.update, avoiding extra conditions in Main.update, and by using clear boolean flags (boss_active, delivering, resting, reprimand) plus timers to ensure that every special state always has a defined exit.

Another frequent issue was repeated boss activation when multiple boxes failed in a row; this was fixed by wrapping boss activation in if not self.boss_active so a new reprimand cannot start until the previous one has finished. Finally, the package spawn logic was adjusted to avoid ending up with no boxes after a series of fails, temporarily simplifying the spawn condition while debugging the boss and truck systems.

Bibliography