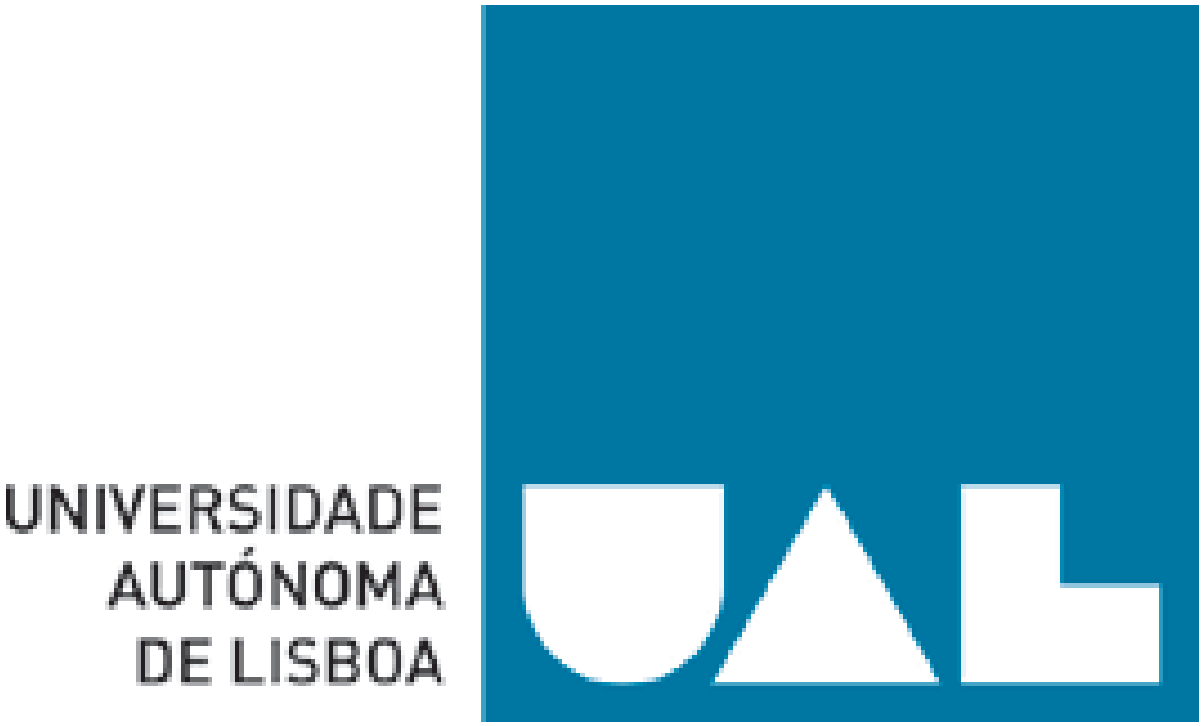
Park Jam PP Project

Programming Paradigms

Christian Isaac Avendaño Tellez

01/06/2025



1. Project Summary

This project for the programming paradigms class features a classic park jam style game. The key components of said project are: A board with cars of varying sizes and colors, a section with 6 different platforms in which the cars that are freed from the board will park, a line of passengers of the same colors as the cars waiting to board the vehicles.
The objective of the game is to free all the cards of the board in a specific order that ensures all the passengers in the line are able to board their assigned vehicles without the platforms getting full.

1.1. Main Goals

Successfully develop a park jam video game with the correct implementation of all the features listed above through object oriented programming.

The game experience must provide fun elements as well as appropriate sound cues, animations and player feedback for it to be a pleasant and entertaining experience.

1.2. Technologies Used

Python: The programming language in which the project will be developed is Python. Thanks to the language combination of simplicity, expressiveness, powerful libraries, and it being a high-level language, Python allows developers to focus on solving problems rather than managing low-level details like memory or boilerplate syntax. These qualities are beneficial when handling a project with real time state updates and the object-oriented modeling of cars, passengers, and game logic.

PyQt6: This framework is crucial due to the fact that it provides an extensive collection of widgets (like QLabel, QPushButton, QVBoxLayout) that make it easy to build the dynamic board, passenger queue, and platform area. Its animation support and the ability to apply Qt Style Sheets make it easy to customize and define the game's aesthetic

Numpy: While ParkJamPP is GUI centric, for the logical side of the board matrix, NumPy was used. By using Numpy we are able to implement a simplified grid.based checks through fast slicing and indexing. Cleaner code and improved performance are other traits the library brought to the project on the logical side.

2. System Architecture and Module Explanations

2.1. Project File Structure

```
ParkJamPP/
├── main.py                  # Punto de entrada del juego
├── assets/                  # Recursos gráficos y sonidos
│   ├── window_background.png # Imagen de fondo de la ventana
│   ├── board_background.png  # Imagen de fondo del tablero
│   ├── cars/                # Imágenes de autos (por color y capacidad)
│   │   ├── orangecar4.png
│   │   ├── orangecar6.png
│   │   ├── orangecar8.png
│   │   ├── orangecar12.png
│   │   ├── redcar4.png
│   │   └── ... (otros colores y capacidades)
│   └── sounds/              # Efectos de sonido para animaciones
│       ├── car_move.wav
│       └── error.wav
├── game/                    # Lógica del juego
│   ├── init.py
│   ├── car.py               # Clase Car y enumeración Direction
│   ├── passenger.py         # Clase Passenger
│   ├── game_controller.py   # Controlador principal, lógica de juego y colisiones
│   └── platform.py          # Gestión de plataformas (estacionamientos)
└── ui/                      # Interfaz gráfica/
├── ├── init.py
└── └── game_window.py       # Ventana principal, dibuja tablero, maneja animaciones y sonidos
```

2.2. Car.py

The car.py module defines the Car class and the Direction enumeration, which together model the fundamental behavior and structure of the vehicles on the game.

Each Car instance stores attributes such as its color, passenger capacity, direction, and head position (row, col). The class computes its logical length based on capacity that then is used to determine occupied grid cells through get_car_cells_static().

The module also provides methods for retrieving the front position (get_front_position()), adjusting for accurate visual placement (get_visual_position()), and generating the appropriate image (QPixmap) based on direction and scale.

Also features the boarded attribute that tracks how many passengers have entered the car, enabling interaction with the platform logic. The use of the Direction enum improves control flow and clarity when calculating movement and rotation.

2.3. Platform.py

On platform.py lies the implementation of the PlatformManager class, which manages the logic of cars that have successfully exited the main board and are waiting to board passengers on the platform. It maintains a list of active platform cars, enforces a maximum platform capacity (capped at 6), and coordinates the boarding process using a reference to a shared PassengerQueue object.

The method add_car_to_platform() tries to immediately board the passengers of matching color upon a car arrival, while board_waiting_passengers() continues the boarding process based on queue state. The system also tracks partial boarding progress via the boarded attribute in each Car, and removes fully boarded cars from the platform once their capacity is met.

2.4. Passenger.py

The passenger.py module defines all essential components for managing the flow of passengers in the game, those being the Passenger class and the PassengerQueue class.

The Passenger class is a simple data structure that encapsulates the color attribute of each passenger, this allows passengers to be matched with cars of the same color during the boarding process.

The PassengerQueue class uses a deque from Python's collections module to implement an efficient FIFO queue system, supporting constant-time addition and removal of passengers. It also provides key methods such as next_passenger() for dequeuing, peek() for viewing the next passenger without removing them, and insert_custom_queue() for initializing a predefined color sequence—used to customize game difficulty and logic.

2.5. Game_controller.py

In the game_controller.py module we find the central logic engine of the project. It manages the state of the 10×10 board using a NumPy matrix (board_matrix) where each cell tracks the ID of the car occupying it. The controller maintains lists of active cars (self.cars), original cars for game reset (self.original_cars), and a dictionary mapping IDs to Car objects.

The method load_initial_data() initializes the board with a predefined set of cars, validating their positions and assigning unique IDs.

The methods that manage the collisions aspect of the car include get_car_cells_static() to determine which cells a car occupies, and is_path_clear() to detect whether a car has an unobstructed path to exit the board. The methods mentioned earlier combined with try_move_car() coordinates the process of removing a car from the board, placing it on the platform, and triggering passenger boarding via the PlatformManager.

The controller also integrates with the PassengerQueue and provides utility functions such as reset_game() and get_car_at_position() to support user interaction and game flow.

UI behaviour and game logic are all bridged in this module as it encapsulates the core rule set, state transitions, and interactions between game entities.

2.6. Game_controller.py

This last module implements the GameWindow class, which defines the graphical user interface using the PyQt6 framework.

Inheriting from QWidget, this class creates and manages all visual elements of the game, including the 10×10 board grid, car widgets, passenger queue display, and platform area. It uses a combination of QVBoxLayout, QGridLayout, and QLabel/QPushButton components to build said interface.

Cars are rendered with dynamic images that are scaled and rotated according to their direction and size, and are placed using their calculated visual positions.

Clicking a car triggers the try_move_car() method, which communicates with the GameController to determine whether the move is valid and then plays the corresponding animation (animate_move() or animate_blocked()).

The platform is updated using update_platforms(), which handles the real time transitions with QPropertyAnimation and QGraphicsOpacityEffect for animation effects.

The passenger queue is refreshed via update_passenger_queue_display(), which synchronizes visual and logical states and triggers animations when passengers board.

3. Technical Decisions

3.1. Implementation of PyQt6

PyQt6 was implemented due to its ability to create a fully custom, animated GUI using native widgets, layout managers, and built-in animation tools like. The framework makes it easy to work with object-oriented design, allowing a clear separation between logic and presentation, effectively making the codebase more maintainable and scalable.

3.2. Car Representation on the Board

Each car is represented as an instance of the Car class, which encapsulates its color, direction, capacity, and head position.

The car’s physical presence on the board is mapped to a NumPy matrix (board_matrix) inside the GameController, where the car's unique ID is written into every cell it occupies, based on its calculated length and direction.

3.3. Collision Logic

Collision logic is resolved using the board matrix, which stores the ID of the car occupying each cell.

When attempting to move a car, the is_path_clear() method calculates the linear path in front of the car based on its direction and checks each cell in that path for occupancy. If any cell contains a non-zero value, the path is considered blocked.

3.4. Passenger Loading and Boarding

Managed through the PassengerQueue and PlatformManager classes. When a car exits the board, it is added to the platform and immediately attempts to board passengers of the same color from the front of the queue.

The system uses a deque to maintain queue order and match passengers in FIFO manner. Each car tracks how many passengers it has boarded (boarded attribute), and once it reaches its capacity, it is marked as done and animated to depart.

4. Difficulties Faced and Solutions

4.1. Parity in Logical and GUI Side

One of the main difficulties faced while developing this project was to correctly pair how the cars were displayed visually in the board matrix to the logical side. When cars faced left and up, the head of the car was not correctly detected by the functions assigning the image to each vehicle, this problem caused problems with the collisions system.

To fix this issue the ID identifier for each car, the numpy library for more precise matrix operations and a board debugging button to see how is the logical data being processed where implemented to find a solution to this error. All these features combined made it easier to tweak the necessary functions to ensure parity.

4.2. Platform Display and Management

One of the most challenging parts of the project was implementing the platform section in a way that was both functional and visually polished. Initially, displaying cars in the platforms caused layout and alignment issues, especially when cars needed to appear dynamically and leave after boarding. To resolve this, it was needed to redesign how each platform slot was constructed using nested QVBoxLayout containers and carefully control the widgets. Other layer of complexity came from triggering animations using QPropertyAnimation and QGraphicsOpacityEffect, which required precise timing for them to show meaningfully. Although not all animations are completely visible in the final product.

4.3. Level Design and Passenger Queue

For the passenger queue, the original idea was to have a random queue generated each time the game was booted. It proved more difficult to implement a random passenger line due to the fixed state of the car positions. Random passengers could make some of the game attempts unsolvable, so the solution was to adapt the passenger class with a method that allowed the making of a custom made queue.

In the aspect of the level design of the board, there was extensive tests to ensure that the board configuration and the line of passengers gave the players a challenging, yet fun and more importantly, solvable experience.

5. Technical Metrics Table

Metric	**Value**
Modules	7
Total Lines of Code	1292
Total Comments in Code	202

6. Project Screenshots