# Adventure World Theme Park Simulation

## Project Report

**COMP1005 Fundamentals of Programming - Postgraduate Assignment**

**Student Name:** Riajul **Assignment:** Adventure World Theme Park Simulation **Semester:** 2, 2025 **Date:** October 16, 2025

## Table of Contents

1. [Overview](#Xc8951b796c7b61cbdd4998c72efbdeed0056e13)
2. [User Guide](#X3a8ade4745be1a19d3791270211caeb887f1cc4)
3. [Traceability Matrix](#X78c5bcbbb6cc6d8dddf9296d1ac3cd3dc9d0177)
4. [Discussion](#X81a060a18a12739ffcbc529f09a725679a256cf)
5. [Showcase](#X78044907923303e194360f13181f0aa1483909c)
6. [Conclusion](#X6b297cc947e0db5526be369b1f0f4ddc67b2415)
7. [Future Work](#X70123913d73623800d57c00e6da9870330ecf47)
8. [References](#Xcb676ea04c4251cf0b78938329be5cf22f38716)

## 1. Overview

### 1.1 Program Purpose

Adventure World is a theme park simulation system that depicts how a recreational facility with rides, self-service agents, and ways for people to wait in line works. The animation of patron behavior and ride operations in the simulation exhibits object-oriented programming techniques, real-time visualization, and statistical analysis.

### 1.2 Implemented Features

The simulation successfully implements the following core features:

**1. Multiple Ride Types**

* Rotating Ferris wheel
* Swinging pirates ship
* Bumper Cars with circular arena movement
* Roller Coaster (Tower Drop) with vertical track animation

**2. Autonomous Patron System**

* Target-based movement using vector mathematics
* State machine implementation (ROAMING, QUEUING, RIDING, LEAVING)
* Collision detection and avoidance
* Patience mechanism leading to park departure
* Random spawn points at designated entrances

**3. Queue Management System**

* FIFO (First-In-First-Out) queue implementation using Python deque
* Capacity enforcement per ride
* Visual queue representation
* State synchronization between patrons and rides

**4. Terrain and Boundaries**

* Park boundary enforcement
* Pathway visualization (horizontal and vertical)
* Entry/exit point system
* Bounding box collision detection for rides

**5. Dual User Interface Modes**

* Interactive mode with user prompts (-i flag)
* Batch mode with CSV configuration files (-f and -p flags)
* Command-line argument parsing using argparse

**6. Simulation Engine**

* Timestep-based discrete event simulation
* step\_change() method for all entities
* 5-timestep patron initialization freeze (per specification)
* Probabilistic patron spawning

**7. Real-time Statistics (Postgraduate Requirement)**

* Dual subplot visualization (park map and statistics graph)
* Time-series tracking of patron states
* Summary statistics output
* Automatic visualization export to PNG

## 2. User Guide

### 2.1 System Requirements

* Python 3.6 or higher
* NumPy library
* Matplotlib library

### 2.2 Installation

# Install required dependencies  
pip install numpy matplotlib  
  
# Or using virtual environment  
python3 -m venv venv  
source venv/bin/activate # Linux/Mac  
pip install numpy matplotlib

### 2.3 Running the Simulation

#### Interactive Mode

Run with user prompts for configuration:

python3 adventureworld.py -i

The program will prompt for:

* Park dimensions (width and height)
* Number of each ride type (Ferris Wheel, Pirate Ship, Bumper Cars, Roller Coaster)
* Initial patron count
* Maximum simulation timesteps

#### Batch Mode

Run with pre-configured CSV files:

python3 adventureworld.py -f map1.csv -p parameters.csv

Where:

* map1.csv defines ride positions and properties
* parameters.csv defines simulation parameters

#### GUI Animation Mode

For code editor like VS Code or Jupyter, force GUI mode:

python3 adventureworld.py -i --gui

### 2.4 Configuration File Formats

**Map File (map.csv):**

ride\_type,x,y,param1,param2,capacity,duration,name  
FerrisWheel,50,150,20,0,8,80,Ferris Wheel  
RollerCoaster,150,150,20,60,6,60,Tower Drop

**Parameter File (parameters.csv):**

parameter,value  
park\_width,200  
park\_height,200  
max\_timesteps,400  
initial\_patrons,10

### 2.5 Output

* **Console:** Shows the real time data with statistics.
* **Visualization:** simulation\_result.png shows the park layout with a graph
* **Statistics:** Number of Patron , ride utilization, queue lengths

## 3. Traceability Matrix

| Feature | Code Reference | Test Method | Test Result | Completion Date |
| --- | --- | --- | --- | --- |
| **1. Base Ride Class** | adventureworld.py:20-122 - class Ride with state management (IDLE, LOADING, RUNNING), bounding boxes, overlap detection | Tested by running all ride types, verifying state transitions | **Pass** | Oct 15, 2025 |
| **2. Ferris Wheel** | adventureworld.py:124-168 - class FerrisWheel(Ride) with rotation animation | Visual verification of gondola rotation in output PNG | **Pass** | Oct 15, 2025 |
| **3. Pirate Ship** | adventureworld.py:171-228 - class PirateShip(Ride) with pendulum motion | Visual verification of swinging motion in output PNG | **Pass** | Oct 15, 2025 |
| **4. Bumper Cars** | adventureworld.py:231-291 - class BumperCars(Ride) with circular movement | Visual verification of cars moving in arena | **Pass** | Oct 15, 2025 |
| **5. Roller Coaster** | adventureworld.py:294-348 - class RollerCoaster(Ride) with vertical movement | Visual verification of car moving up/down track | **Pass** | Oct 15, 2025 |
| **6. Patron Class** | adventureworld.py:351-478 - class Patron with state machine, movement algorithms | Run simulation, verify patrons move and change states | **Pass** | Oct 15, 2025 |
| **7. Target-Based Movement** | adventureworld.py:397-459 - \_roam() method with vector calculations | Tested by observing patron pathfinding to rides | **Pass** | Oct 16, 2025 |
| **8. Collision Detection** | adventureworld.py:58-70 - is\_in\_bounds() and overlaps() methods | Verified patrons don’t enter ride bounding boxes | **Pass** | Oct 15, 2025 |
| **9. Queue System** | adventureworld.py:72-76, 88-108 - add\_to\_queue() and deque operations | Checked queue formation in console output and PNG | **Pass** | Oct 16, 2025 |
| **10. Theme Park Manager** | adventureworld.py:481-644 - class ThemePark with simulation loop | Run complete simulation, verify coordination | **Pass** | Oct 15, 2025 |
| **11. Statistics Tracking** | adventureworld.py:565-575, 610-629 - Statistics history and plotting | Verified graphs show correct data trends | **Pass** | Oct 15, 2025 |
| **12. File Loading** | adventureworld.py:647-724 - CSV parsers for maps and parameters | Tested batch mode with various CSV files | **Pass** | Oct 15, 2025 |
| **13. Interactive Mode** | adventureworld.py:727-783 - User input prompts and validation | Tested with various input values | **Pass** | Oct 15, 2025 |
| **14. Batch Mode** | adventureworld.py:786-812 - File-based configuration | Tested with map1.csv and parameters.csv | **Pass** | Oct 15, 2025 |
| **15. Command-Line Interface** | adventureworld.py:876-915 - argparse implementation with -i, -f, -p, –gui flags | Tested all flag combinations | **Pass** | Oct 16, 2025 |
| **16. Headless Detection** | adventureworld.py:821-856 - Backend detection and PNG export | Tested in terminal and VS Code environments | **Pass** | Oct 16, 2025 |
| **17. 5-Timestep Freeze** | adventureworld.py:375-384 - frozen\_time check in patron step | Verified patrons don’t move for first 5 steps | **Pass** | Oct 15, 2025 |

## 4. Discussion

### 4.1 System Architecture

The Adventure World simulation is followed by an object-oriented design pattern. There are three primary classes:

#### 4.1.1 Ride Hierarchy

The Ride base class reflects common functional for all attractions, including:

* State management (IDLE → LOADING → RUNNING → IDLE cycle)
* Bounding box calculations to detect collision
* Python’s collections.deque for managing queue
* Abstract \_calculate\_angle() method for indiviudal subclass animations.

Four concrete ride classes is dervied class of Ride:

* **FerrisWheel:** Using trigonometry with multiple gondolas position for 360 degree contnuous rotation.
* **PirateShip:** for realistic pendulum swinging, used sinusoidal motion.
* **BumperCars:** Features circular movement patterns for multiple cars in an arena
* **RollerCoaster:** for tower drop simulation implemented vertical oscillation

#### 4.1.2 Patron Agent System

The Patron class implements agent behavior through:

* **State Machine:** Four types of patron controlling (ROAMING, QUEUING, RIDING, LEAVING) behavior
* **Target-Based Movement:** Uses vector mathematics for smooth navigation toward rides or exit points
* **Collision Avoidance:** Verify ThemePark.is\_valid\_position() before individual movement step
* **Patience Mechanism:** Accumulated patience counter triggers park departure after threshold

#### 4.1.3 Theme Park Manager

The ThemePark class works as the simulation coordinator:

* Manages collections of rides and visitors
* Runs the main simulation cycle using the step() function
* Includes validation methods to check positions and detect overlaps
* Collects and updates performance statistics at every time interval
* Oversees graphical display using matplotlib for visualization

### 4.2 UML Class Diagram

┌─────────────────────────┐  
│ ThemePark │  
├─────────────────────────┤  
│ - width: int │  
│ - height: int │  
│ - rides: List[Ride] │  
│ - patrons: List[Patron] │  
│ - exits: List[Tuple] │  
│ - timestep: int │  
│ - stats\_history: Dict │  
├─────────────────────────┤  
│ + add\_ride() │  
│ + add\_patron() │  
│ + step() │  
│ + plot() │  
│ + is\_valid\_position() │  
└───────────┬─────────────┘  
 │ contains  
 │  
 ┌───────┴────────┐  
 │ │  
 ▼ ▼  
┌─────────┐ ┌──────────┐  
│ Ride │ │ Patron │  
├─────────┤ ├──────────┤  
│ - x,y │ │ - x,y │  
│ - state │ │ - state │  
│ - queue │ │ - speed │  
├─────────┤ ├──────────┤  
│+ plot() │ │+ \_roam() │  
│+ step\_ │ │+ plot() │  
│ change │ └──────────┘  
└────┬────┘  
 │ inherits  
 │  
 ┌──┴───┬────────┬──────────┐  
 ▼ ▼ ▼ ▼  
┌────┐ ┌────┐ ┌────┐ ┌─────────┐  
│FW │ │PS │ │BC │ │RC │  
└────┘ └────┘ └────┘ └─────────┘

### 4.3 Key Design Decisions

#### 4.3.1 Continuous Animation

Every ride stays moving, even when no one is on it, to keep things interesting and show that the simulation is still going. This is done by changing self.angle every time step\_change() is called, whether or whether the ride is full.

#### 4.3.2 Vector-Based Movement

Patrons use normalized direction vectors for smooth movement:

dx = target\_x - self.x  
dy = target\_y - self.y  
dist = sqrt(dx² + dy²)  
new\_x = x + speed \* (dx / dist)  
new\_y = y + speed \* (dy / dist)

This provides natural-looking navigation compared to grid-based movement.

#### 4.3.3 Deque for Queue Management

Python’s collections.deque was chosen for O(1) append and popleft operations, essential for efficient FIFO queue processing:

self.queue = deque()  
self.queue.append(patron) # O(1)  
patron = self.queue.popleft() # O(1)

#### 4.3.4 Probabilistic Spawning

New patrons spawn with 20% probability per timestep (when below capacity), creating realistic variable arrival rates rather than deterministic spawning.

### 4.4 Implementation Challenges and Solutions

**Challenge 1: Patrons Not Joining Queues**

* **Problem:** At first, patrons rarely joined any rides because the probability was set too low (2%) and the distance limit (15 units) was too short. As a result, almost no rides were used during the simulation.
* **Solution:** The probability was raised to 8%, and the distance threshold was extended to 35 units. This adjustment also considered the ride’s bounding box area, making it easier for patrons to detect and join nearby rides.

**Challenge 2: Headless Environment Detection**

* **Problem:** When running in a headless setup, matplotlib automatically switched to a non-interactive backend, which caused the visual display to fail quietly.
* **Solution:** A detection system was added to check the backend in use, along with a –gui flag that forces the TkAgg backend when a graphical interface is available.

**Challenge 3: Animation Not Running**

* **Problem:** In non-interactive backends, plt.show() doesn’t block execution, preventing the animation from running as expected.
* **Solution:** The issue was fixed by implementing two different execution modes — a manual time-step loop for headless environments and FuncAnimation for GUI-based execution.

## 5. Showcase

### 5.1 Introduction

Three alternative configurations were made and tested to show how the simulation works and changes when conditions change. Each scenario focuses on testing certain parts of the system:

**Scenario 1: Standard Park**

* Setup: map1.csv with parameters.csv
* Goal: Serve as a baseline run using balanced settings
* Focus: Ensure all essential components work as intended

**Scenario 2: Extended Park**

* Setup: map2.csv with parameters2.csv
* Goal: Assess performance and scalability with a larger park layout and longer runtime
* Focus: Observe how the system handles greater complexity and workload

**Scenario 3: Interactive Configuration**

* Setup: User-defined through the interactive configuration mode
* Goal: Demonstrate the system’s flexibility in accepting user inputs
* Focus: Show how easily the simulation can adapt to custom settings

All scenarios were executed using the command:

python3 adventureworld.py -f <map\_file> -p <param\_file>

Statistics were recorded from console output and visualization was captured in simulation\_result.png.

### 5.2 Scenario 1: Standard Park

#### Configuration

**Command:**

python3 adventureworld.py -f map1.csv -p parameters.csv

**Parameters:**

* Park dimensions: 200 × 200 units
* Rides: 4 (Ferris Wheel, Pirate Ship, Bumper Cars, Tower Drop)
* Initial patrons: 10
* Maximum timesteps: 400
* Duration: ~30 seconds

#### Results

**Console Output Summary:**

Total timesteps: 400  
Total patrons entered: 33  
Total patrons left: 3  
Patrons still in park: 30  
Total rides taken: 69  
  
Ride Statistics:  
 Ferris Wheel: 8 riders, queue: 4, state: RUNNING  
 Tower Drop: 14 riders, queue: 2, state: RUNNING  
 Bumper Cars: 19 riders, queue: 4, state: RUNNING  
 Pirate Ship: 28 riders, queue: 1, state: RUNNING

**Visualization Analysis:**

* The four rides appear clearly at the park’s corners — around coordinates (50,150), (150,150), (50,50), and (150,50).
* Patrons, represented by small colored dots, are scattered across the park area.
* Queues are shown as colored squares positioned near each ride.
* The statistics chart displays the following trends:
  + Blue line (Total Patrons): Rises from about 10 to nearly 30 within the first 100 timesteps, then levels off.
  + Orange line (In Queue): Moves up and down between roughly 3 and 13 patrons.
  + Green line (On Rides): Fluctuates within the range of 8 to 14 patrons.

**Discussion:** This test shows that the main sections of the simulation are doing what they should be doing. The Pirate Ship ride was the most popular, with 28 people riding it. This ride was undoubtedly popular because it could hold more people (10 clients) and it lasted a long time (50 timesteps). In the end, the total number of customers stayed about 30, which suggests that there were about the same number of new customers as there were people who left after waiting. The number of people in line kept at a respectable level, usually between one and four. This suggested that the rides were well-managed in terms of time and space. The fact that the queuing and riding numbers are both stable and not 0 suggests that customers are using the system correctly by finding rides, waiting in line, and finishing their rides as anticipated.

### 5.3 Scenario 2: Extended Park

#### Configuration

**Command:**

python3 adventureworld.py -f map2.csv -p parameters2.csv

**Parameters:**

* Park dimensions: 220 × 220 units
* Rides: 6 (2 Ferris Wheels, 1 Tower Drop, 2 Bumper Cars, 1 Pirate Ship)
* Initial patrons: 15
* Maximum timesteps: 600
* Duration: ~45 seconds

#### Results

**Console Output Summary:**

Total timesteps: 600  
Total patrons entered: 48  
Total patrons left: 12  
Patrons still in park: 36  
Total rides taken: 127  
  
Ride Statistics:  
 Sky Wheel: 18 riders, queue: 3, state: RUNNING  
 Wonder Wheel: 22 riders, queue: 2, state: RUNNING  
 Drop Tower: 21 riders, queue: 1, state: RUNNING  
 Crash Arena: 28 riders, queue: 4, state: RUNNING  
 Bump Zone: 24 riders, queue: 3, state: RUNNING  
 Sea Storm: 14 riders, queue: 2, state: IDLE

**Discussion:** This long test case shows that the system can handle additional work without any difficulties. The number of completed ride cycles went boosted by about 84% (from 69 to 127) when the park featured 50% more rides and the simulation lasted for 50% longer. There was no clear drop in performance. Each of the two Ferris Wheels could hold 18 and 22 people, which kept demand steady by preventing any one ride from getting too crowded.  
The way the lines are managed is still the same, and the biggest line only has four people in it. This shows that the scaling maintained the load balanced. Because there are usually timing problems between ride cycles, it's normal for one ride to end in an IDLE state at the end of the simulation.

### 5.4 Scenario 3: Interactive Configuration

#### Configuration

**Command:**

python3 adventureworld.py -i

**User Inputs:**

* Park width: 200
* Park height: 200
* Ferris Wheels: 1
* Pirate Ships: 1
* Bumper Cars: 1
* Roller Coasters: 1
* Initial patrons: 10
* Max timesteps: 400

#### Results

**Console Output Summary:**

Total timesteps: 400  
Total patrons entered: 31  
Total patrons left: 2  
Patrons still in park: 29  
Total rides taken: 74  
  
Ride Statistics:  
 Ferris Wheel: 16 riders, queue: 2, state: RUNNING  
 Pirate Ship: 22 riders, queue: 1, state: RUNNING  
 Bumper Cars: 26 riders, queue: 3, state: RUNNING  
 Roller Coaster: 10 riders, queue: 0, state: RUNNING

**Discussion:** The interactive mode displays how well the user interface works in real life. The scenario was like Scenario 1, but the results were different (74 rides instead of 69) because people came and went at random. The command-line interface correctly checked what users entered and created the ride objects on the fly. Users can alter parameters and try out different setups with this way without having to edit the code or CSV files. This is ideal for testing and trying things out.

### 5.5 Comparative Analysis

| Metric | Scenario 1 | Scenario 2 | Scenario 3 |
| --- | --- | --- | --- |
| Rides | 4 | 6 | 4 |
| Timesteps | 400 | 600 | 400 |
| Total Rides Taken | 69 | 127 | 74 |
| Avg Rides/Attraction | 17.3 | 21.2 | 18.5 |
| Max Patron Count | 30 | 36 | 29 |
| Throughput (rides/100 steps) | 17.3 | 21.2 | 18.5 |

**Key Observations:**

1. **Linearity:** Ride usage tends to grow roughly in proportion to the number of timesteps (127/600 ≈ 69/400).
2. **Capacity:** Adding more rides increases overall ride completions without significantly raising the peak number of patrons in the park.
3. **Stochasticity:** Even with similar setups, results varied by 5–7% due to the probabilistic nature of patron arrival and movement.
4. **Stability:** Across all scenarios, patron numbers remained steady, with no uncontrolled growth observed.

## 6. Conclusion

The Adventure World theme park simulation is a great example of how to integrate object-oriented programming, discrete event simulation, and real-time data visualization in real life. All seven prerequisites for the assignment were successfully met and checked:  
Things that have been done:  
1. ✅ Four different sorts of rides, each with its own set of animations.  
2. ✅ Autonomous patron agents that act based on the state.  
3. ✅ Using the right data structures to manage queues effectively.  
4. ✅ Park terrain with paths that are easy to see and collision detection.  
5. Two ways to use the interface: interactive and batch.  
6. ✅ A fully working simulation engine that moves the time forward correctly.  
7. ✅ Real-time statistics shown in subplot format, which meets the needs of graduate students.  
The simulation is a realistic computer model of a theme park that shows how basic programming skills may be used to imitate and study complicated real-world systems.

7. Work to be done in the future  
There are a number of ways to make the simulation more advanced and more like how parks work in real life:  
7.1 Better Pathfinding  
Current: Patrons use simple vector math to move toward their targets. Improvement: Add the A\* pathfinding algorithm so that users may smartly go past barriers and congested regions. This would let them pick the best routes based on both distance and traffic.  
7.2 Loading Dynamic Terrain from Files  
Now: The ThemePark.plot() method has hard-coded pathways and park features. Improvement: Load park features like benches, barriers, or food stalls from CSV files outside of the program. For example: terrain\_type,x,y,width,height barrier,100,50,10,100 bench,75,75,5,5  
This would let you change the layouts of the parks without modifying the source code.  
7.3 What kinds of rides do patrons like?  
Now: Patrons choose rides at random from the ones that are available. Improvement: Give clients weighted preferences that show how much they like each attraction based on their own interests:  
self.preferences = { 'FerrisWheel': random.uniform(0.5, 1.5), 'PirateShip': random.uniform(0.5, 1.5), # ... }  
This is a more realistic way to make decisions, where some rides are better than others.

## 8. References

### Course Materials

1. COMP1005 Lecture Slides - “Object-Oriented Programming in Python” (Weeks 4-6, 2025)
2. COMP1005 Practical Test 3 - “Pirate Ship Animation” (provided code basis for ride movement patterns)
3. COMP1005 Practical Exercises - “Pet Shelter Queue Management” (informed queue implementation)

### Assignment Documentation

1. COMP1005 Assignment Specification v1.0 - “Adventure World” (Semester 2, 2025)

### External Documentation

1. Python Software Foundation. (2025). *Python 3.12 Documentation*. Retrieved from https://docs.python.org/3/
2. Hunter, J. D. (2007). “Matplotlib: A 2D Graphics Environment”. *Computing in Science & Engineering*, 9(3), 90-95.
3. Harris, C. R., et al. (2020). “Array programming with NumPy”. *Nature*, 585, 357-362.

### Style Guides

1. van Rossum, G., Warsaw, B., & Coghlan, N. (2001). *PEP 8 – Style Guide for Python Code*. Python.org. Retrieved from https://www.python.org/dev/peps/pep-0008/

**Word Count:** Approximately 4,200 words **Page Count:** Approximately 11 pages (with diagrams and tables)

**Author:** Riajul **Student ID:** [Your Student ID] **Date Submitted:** October 16, 2025

## Appendix A: Code Snippet Examples

### A.1 Ride State Transition Logic

def step\_change(self):  
 self.time\_counter += 1  
 self.angle = self.\_calculate\_angle()  
  
 if self.state == "IDLE" and len(self.queue) > 0:  
 self.state = "LOADING"  
 elif self.state == "LOADING":  
 while len(self.riders) < self.capacity and len(self.queue) > 0:  
 patron = self.queue.popleft()  
 self.riders.append(patron)  
 patron.state = "RIDING"  
 self.total\_riders += 1  
 if len(self.riders) > 0:  
 self.state = "RUNNING"  
 self.time\_counter = 0  
 elif self.state == "RUNNING":  
 if self.time\_counter >= self.duration:  
 for rider in self.riders:  
 rider.state = "ROAMING"  
 rider.target\_ride = None  
 self.riders = []  
 self.state = "IDLE"

### A.2 Patron Movement Algorithm

def \_roam(self):  
 if self.target\_ride is None and np.random.random() < 0.08:  
 available\_rides = [r for r in self.park.rides if len(r.queue) < 8]  
 if available\_rides:  
 self.target\_ride = np.random.choice(available\_rides)  
 self.target\_x = self.target\_ride.x  
 self.target\_y = self.target\_ride.y  
  
 if self.target\_ride is not None:  
 dx = self.target\_x - self.x  
 dy = self.target\_y - self.y  
 dist = np.sqrt(dx\*\*2 + dy\*\*2)  
  
 if dist < 35:  
 self.target\_ride.add\_to\_queue(self)  
 self.rides\_taken += 1  
 self.patience = 0  
 return  
  
 new\_x = self.x + self.speed \* (dx / dist)  
 new\_y = self.y + self.speed \* (dy / dist)  
  
 if self.park.is\_valid\_position(new\_x, new\_y):  
 self.x = new\_x  
 self.y = new\_y

**END OF REPORT**