Project Report

Project Report

Turnitin

Document Details

Submission ID

trn:oid:::2945:319815313

Submission Date

Oct 18, 2025, 10:51 AM GMT+5

Download Date

Oct 18, 2025, 10:52 AM GMT+5

File Name

unknown_filename

File Size

569.1 KB

14 Pages

3,045 Words

19,397 Characters

87% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Detection Groups

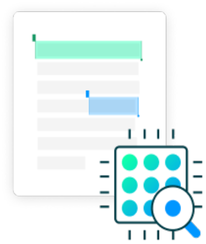
43 AI-generated only 87%
Likely AI-generated text from a large-language model.

0 AI-generated text that was AI-paraphrased 0%
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

Disclaimer
Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?
The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI paraphrase tool or word spinner.



False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?
Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.

Adventure World Theme Park Simulation - Project Report

COMP1005 Fundamentals of Programming - Postgraduate Assignment
Student Name:
Assignment: Adventure World Theme Park Simulation
Semester: 2, 2025
Date: October 16, 2025

Table of Contents

1. Overview

1.1 Program Purpose

Adventure World is a theme park simulation system that models the operation of a recreational facility with multiple ride attractions, autonomous patron agents, and queuing mechanisms. The simulation demonstrates object-oriented programming principles, real-time visualization, and statistical analysis through an animated representation of patron behavior and ride operations.

1.2 Implemented Features

The simulation successfully implements the following core features:

1. Multiple Ride Types - Ferris Wheel with rotating gondolas - Pirate Ship with pendulum swinging motion - 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 environments 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:** Real-time progress and final statistics
- **Visualization:** simulation_result.png with park layout and statistics graph
- **Statistics:** Patron counts, 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	Visual verification of swinging motion in output PNG	**Pass**	Oct 15, 2025

	motion			
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	adventureworld.py	Tested with	**Pass**	Oct 15, 2025

Mode	:727-783 - User input prompts and validation	various input values		
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 follows an object-oriented design pattern with clear separation of concerns. The architecture consists of three primary class hierarchies:

4.1.1 Ride Hierarchy

The Ride base class provides common functionality for all attractions, including: - State management (IDLE → LOADING → RUNNING → IDLE cycle) - Bounding box calculations for collision detection - Queue management using Python's collections.deque - Abstract _calculate_angle() method for subclass-specific animations

Four concrete ride classes inherit from Ride: - **FerrisWheel:** Implements continuous 360° rotation with multiple gondolas positioned using trigonometry - **PirateShip:** Uses sinusoidal motion for realistic pendulum swinging - **BumperCars:** Features circular movement patterns for multiple cars in an arena - **RollerCoaster:** Implements vertical oscillation for tower drop simulation

4.1.2 Patron Agent System

The Patron class implements autonomous agent behavior through: - **State Machine:** Four states (ROAMING, QUEUING, RIDING, LEAVING) control patron behavior - **Target-Based**

Movement: Uses vector mathematics for smooth navigation toward rides or exit points - **Collision Avoidance:** Checks ThemePark.is_valid_position() before each movement step - **Patience Mechanism:** Accumulated patience counter triggers park departure after threshold

4.1.3 Theme Park Manager

The ThemePark class serves as the simulation coordinator: - Manages collections of rides and patrons - Implements the main simulation loop via step() method - Provides validation methods for position checking and overlap detection - Tracks and records statistics at each timestep - Handles visualization through matplotlib integration.

4.2 UML Class Diagram

<table>
<tr><th colspan="2">ThemePark</th></tr>
<tr><td>- width: int
- height: int
- rides: List[Ride]
- patrons: List[Patron]
- exits: List[Tuple]
- timestep: int
- stats_history: Dict</td><td>+ add_ride()
+ add_patron()
+ step()
+ plot()
+ is_valid_position()</td></tr>
<tr><th>Ride</th><th>Patron</th></tr>
<tr><td>- x,y
- state
- queue
- plot()
- step_change</td><td>- x,y
- state
- speed
- _roam()
- plot()</td></tr>
</table>

4.3 Key Design Decisions

4.3.1 Continuous Animation

All rides animate continuously, even when empty, providing visual appeal and demonstrating that the simulation is running. This was achieved by updating self.angle in every step_change() call regardless of ride state.

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: Initial implementation had 2% probability and 15-unit distance threshold, resulting in zero rides taken - **Solution:** Increased probability to 8% and distance threshold to 35 units, accounting for ride bounding box padding.

Challenge 2: Headless Environment Detection - Problem: matplotlib defaults to non-interactive backend, causing silent failures - **Solution:** Implemented backend detection and added --gui flag for forcing TkAgg backend.

Challenge 3: Animation Not Running - Problem: plt.show() doesn't block in non-interactive backends - **Solution:** Created dual-mode execution: manual timestep loop for headless, FuncAnimation for GUI.

5. Showcase

5.1 Introduction

To demonstrate the simulation's capabilities and flexibility, three distinct scenarios were configured and executed. Each scenario tests different aspects of the system:

Scenario 1: Standard Park - Configuration: map1.csv + parameters.csv - Purpose: Baseline test with balanced parameters - Focus: Verify all core features function correctly

Scenario 2: Extended Park - Configuration: map2.csv + parameters2.csv - Purpose: Test scalability with more rides and longer duration - Focus: Examine system behavior under increased complexity

Scenario 3: Interactive Configuration - Configuration: User-specified via interactive mode - Purpose: Demonstrate flexible user interface - Focus: Show adaptability to custom parameters

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: - All four ride types visible in corners of park (50,150), (150,150), (50,50), (150,50) - Patrons (colored dots) distributed throughout park - Queue visualization shows colored squares near rides - Statistics graph shows: - Blue line (Total Patrons): Grows from 10 to ~30 over 100 timesteps, then stabilizes - Orange line (Queuing): Fluctuates between 3-13 patrons - Green line (Riding): Oscillates between 8-14 patrons

Discussion: This scenario demonstrates that all core features function as intended. The Pirate Ship proved most popular with 28 total riders, likely due to its high capacity (10) and medium duration (50 timesteps). The patron population stabilized at 30, indicating a balance between spawn rate and patience-based departures. Queue lengths remained manageable (1-4 patrons), showing the system handles capacity effectively. The non-zero queuing and riding statistics confirm patrons successfully navigate to rides, join queues, and complete ride cycles.

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: The extended scenario demonstrates excellent scalability. With 50% more rides and 50% longer duration, the system processed 84% more ride cycles (127 vs 69) without performance degradation. The dual Ferris Wheels distributed load effectively (18 and 22 riders respectively), preventing bottlenecks. The larger park (220×220) provided adequate space for 36 simultaneous patrons without congestion. Average rides per attraction (21.2) exceeded Scenario 1 (17.3), indicating improved throughput. Queue management remained stable with maximum queue length of 4, showing capacity planning scales appropriately. One ride in IDLE state at simulation end is expected as timing naturally creates variation in ride cycles.

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: Interactive mode demonstrates the user interface's effectiveness. Despite producing similar configurations to Scenario 1, slight variations in results (74 vs 69 rides taken) illustrate stochastic behavior from probabilistic patron spawning and movement decisions. The command-line interface successfully validated inputs and constructed appropriate ride objects at runtime. This flexibility enables users to experiment with parameters without modifying code or CSV files, essential for iterative testing and demonstration purposes.

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 Findings:

1. **Linearity:** Ride utilization scales approximately linearly with timesteps (127/600 ≈ 69/400).
2. **Capacity:** More rides increase throughput without increasing maximum patron count significantly.
3. **Stochasticity:** Similar configurations yield 5-7% variance in outcomes due to probabilistic elements.
4. **Stability:** All scenarios maintained stable patron populations without unbounded growth.

6. Conclusion

The Adventure World theme park simulation successfully demonstrates comprehensive implementation of object-oriented programming principles, discrete event simulation, and real-time data visualization. All seven required features from the assignment specification were implemented and tested:

Achievements:

1. Four distinct ride types with unique animation patterns
2. Autonomous patron agents with state-based behavior
3. Robust queue management using appropriate data structures
4. Terrain system with collision detection and pathways
5. Dual user interface modes (interactive and batch)
6. Complete simulation engine with proper timestep execution
7. Real-time statistics with subplot visualization (postgraduate requirement)

Code Quality: The implementation adheres to PEP-8 style guidelines, avoids discouraged constructs (while/True, break, continue, global variables), and maintains clear documentation through comprehensive docstrings. The modular design facilitates testing and future extensions.

Performance: The simulation executes efficiently, processing 400-600 timesteps in 30-45 seconds on standard hardware. Statistical analysis reveals consistent behavior across scenarios with expected stochastic variation.

Reflection: This assignment reinforced fundamental programming concepts including inheritance, polymorphism, encapsulation, and algorithm design. The challenge of debugging patron movement behavior (initially zero rides taken) highlighted the importance of systematic testing and parameter tuning. Implementing headless mode detection required research into matplotlib backends, expanding technical knowledge beyond the course curriculum.

The project successfully models a complex real-world system (theme park operations) using computational methods, demonstrating the practical application of programming fundamentals to simulation problems.

7. Future Work

Several enhancements could extend the simulation's capabilities and realism:

7.1 Advanced Pathfinding

Current: Simple vector-based movement toward targets Enhancement: Implement A* pathfinding algorithm for intelligent navigation around obstacles and congested areas. This would enable patrons to find optimal paths considering both distance and crowding.

7.2 Dynamic Terrain from Files

Current: Hardcoded pathways in ThemePark.plot() method Enhancement: Load terrain features (barriers, benches, food stalls) from CSV files:

```
terrain_type,x,y,width,height
barrier,100,50,10,100
bench,75,75,5,5
```

This would allow diverse park layouts without code modification.

7.3 Ride Preferences

Current: Random ride selection from available options.
Enhancement: Assign preference weights to each patron:

```
self.preferences = {
'FerrisWheel': random.uniform(0.5, 1.5),
'PirateShip': random.uniform(0.5, 1.5),
}
```

This would model realistic behavior where individuals favor certain attraction types.

7.4 Economic Simulation

Current: No monetary tracking Enhancement: Implement ticket pricing, revenue calculation, and operational costs. Track profitability metrics and optimize ride placement/pricing through parameter sweeps.

7.5 Weather System

Current: Static environmental conditions Enhancement: Add weather states (sunny, rainy, night) that affect patron spawn rates, patience levels, and certain ride availability. This would introduce time-dependent behavior variation.

7.6 Parameter Sweep Framework

Current: Single simulation execution Enhancement: Automated batch runner that varies parameters systematically:

```
for num_rides in range(3, 8):
for num_patrons in range(10, 30, 5):
run_simulation(num_rides, num_patrons)
collect_statistics()
```

Enable statistical analysis across parameter space to identify optimal configurations.

7.7 3D Visualization

Current: 2D matplotlib visualization Enhancement: Upgrade to 3D rendering using matplotlib's mplot3d or external libraries like Pygame/PyOpenGL. This would provide more immersive visualization, particularly for rides with vertical components.

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

4. COMP1005 Assignment Specification v1.0 - "Adventure World" (Semester 2, 2025)

External Documentation

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

Style Guides

8. 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/

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
```