System Requirements Specification Index

For

Game Development Utility System

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

TABLE OF CONTENTS

Game Development Utility System

System Requirements Specification

1 Project Abstract

Pixel Perfect Studios is developing a new game engine and needs a utility system for data processing and game mechanics. This assignment focuses on implementing lambda functions to handle game-related calculations, player data transformations, and entity filtering for efficient game operations. Lambda functions will provide concise, reusable operations for common game development tasks.

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. System must utilize lambda functions for game data processing 2. Code must demonstrate proper lambda function syntax and usage 3. Functions must be called with correct arguments 4. System must perform operations on Player statistics and inventories, Game entity properties and filtering, Score calculations and leaderboards

3 Constraints

3.1 Lambda Function Requirements

1. Lambda Syntax:
    - Must use the keyword `lambda` followed by parameters and expression
    - Must be single-expression functions
    - Example: `lambda player: player["health"] * 2`

2. Usage Contexts:
    - Must use lambda functions with built-in functions (map, filter, sorted)
    - Must use lambda functions as function arguments
    - Must assign at least one lambda function to a variable
    - Must use at least one lambda function in a list comprehension
3. Parameter Types:
    - Must demonstrate lambda functions with single parameter
    - Must demonstrate lambda functions with multiple parameters
    - Must demonstrate lambda functions that operate on different data types
4. Return Values:
    - Lambda functions must return appropriate data types
    - Some lambda functions must return boolean values for entity filtering
    - Some lambda functions must return numeric values for game calculations

3.2 Input Requirements

1. Player Data:
    - Input list must contain at least 5 player dictionaries
    - Each player must have attributes: name, level, health, mana, score
    - Example: `{"name": "Wizard1", "level": 5, "health": 80, "mana": 100, "score": 2500}`
2. Game Entities:
    - Input list must contain at least 8 game entity dictionaries
    - Each entity must have: id, type, position_x, position_y, active
    - Example: `{"id": "E001", "type": "enemy", "position_x": 100, "position_y": 200, "active": True}`
3. Item Inventory:
    - Input list must contain at least 6 item dictionaries
    - Each item must have: name, type, value, rarity, equipped

- Example: `{"name": "Magic Sword", "type": "weapon", "value": 500, "rarity": "rare", "equipped": False}`

4. Game Coordinates:
    - Input list must contain at least 5 coordinate tuples
    - Example: `[(10, 20), (50, 60), (30, 40), (70, 80), (90, 10)]`

3.3 Operations Constraints

1. Player Transformations:
    - Must use lambda with map() to transform player statistics
    - Must convert map result to list to display values
    - Example: `list(map(lambda p: p["health"] * 1.5, players))`
2. Entity Filtering:
    - Must use lambda with filter() to select specific entities
    - Must demonstrate filtering with different criteria
    - Example: `list(filter(lambda e: e["type"] == "enemy" and e["active"], entities))`
3. Item Sorting:
    - Must use lambda with sorted() to order items
    - Must demonstrate custom sorting keys
    - Example: `sorted(inventory, key=lambda item: item["value"])`
4. Game Calculations:
    - Must demonstrate using lambda functions for game mechanics
    - Must show calculating distances, damage, or other game values
    - Example: `lambda pos1, pos2: ((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)**0.5`

3.4 Output Constraints

1. Display Format:
    - Each operation result must have a descriptive game-related label
    - Results must be formatted for readability
    - Must include before and after data states

3. Output Sections:
    - Player transformation results
    - Game entity filtering results
    - Item sorting results
    - Game mechanic calculation results

4. TEMPLATE CODE STRUCTURE:

1. Game Data Preparation:
    - `prepare_player_data()` - creates list of players for processing
    - `prepare_entity_data()` - creates list of game entities
    - `prepare_inventory_data()` - creates list of game items
    - `prepare_coordinate_data()` - creates list of game world coordinates

2. Lambda Function Operations:
    - `demonstrate_player_transformations()` - shows lambda functions for player data
    - `demonstrate_entity_filtering()` - shows lambda functions for filtering entities
    - `demonstrate_item_sorting()` - shows lambda functions for inventory management
    - `demonstrate_game_calculations()` - shows lambda functions for game mechanics

3. Advanced Operations:
    - `demonstrate_ability_system()` - shows lambda functions for game abilities
    - `demonstrate_combat_system()` - shows lambda functions for damage calculations
    - `demonstrate_level_system()` - shows lambda functions for experience and leveling

4. Main Program Function:
    - `main()` - main program function executing all demonstrations
    - Must execute each demonstration function
    - Must present results in structured format

5. Detailed Function Structure:

5.1 Data Preparation Functions

1. Write a Python function to prepare player data for lambda function processing

Define: **prepare_player_data()**

This function should:

- Create and return a list of at least 5 player dictionaries
- Each player dictionary must contain exactly these keys: "name", "level", "health", "mana", "score"
- Use varied realistic values for each player
- Include different player types (e.g., "Wizard1", "Warrior2", "Archer3", "Healer4", "Tank5")
- Ensure level values range from 1-10, health from 50-200, mana from 20-150, score from 1000-5000
- Return the complete list for use in lambda function demonstrations
- Example player: `{"name": "Wizard1", "level": 5, "health": 80, "mana": 100, "score": 2500}`

2. Write a Python function to prepare game entity data for lambda function processing.

Define: **prepare_entity_data()**

This function should:

- Create and return a list of at least 8 game entity dictionaries
- Each entity dictionary must contain exactly these keys: "id", "type", "position_x", "position_y", "active"
- Include different entity types: "enemy", "npc", "item"
- Use realistic position coordinates (x and y values between 50-300)
- Mix of active (True) and inactive (False) entities
- Use descriptive IDs like "E001", "E002", etc.
- Include at least 3 enemies, 2 npcs, and 2 items in the list
- Return the complete list for filtering and processing demonstrations
- Example entity: `{"id": "E001", "type": "enemy", "position_x": 100, "position_y": 200, "active": True}`

3. Write a Python function to prepare inventory data for lambda function processing.

Define: **prepare_inventory_data()**

This function should:

- Create and return a list of at least 6 item dictionaries
- Each item dictionary must contain exactly these keys: "name", "type", "value", "rarity", "equipped"
- Include different item types: "weapon", "armor", "consumable"
- Use rarity levels: "common", "uncommon", "rare", "epic", "legendary"
- Include mix of equipped (True) and unequipped (False) items
- Value should range from 50-1000 gold
- Use descriptive item names like "Magic Sword", "Dragon Shield", "Health Potion"
- Return the complete list for sorting and value calculations
- Example item: {"name": "Magic Sword", "type": "weapon", "value": 500, "rarity": "rare", "equipped": False}

4. Write a Python function to prepare coordinate data for lambda function processing.

Define: **prepare_coordinate_data()**

This function should:

- Create and return a list of at least 5 coordinate tuples

- Each coordinate should be a tuple with exactly 2 values (x, y)
- Use varied coordinate values for distance calculations
- Coordinates should represent game world positions
- Include both positive coordinates for realistic game positioning
- Return tuples, not lists: use (x, y) format
- Example coordinates: `[(10, 20), (50, 60), (30, 40), (70, 80), (90, 10)]`

5.2 Lambda Function Demonstration Functions

5. Write a Python function to demonstrate player transformations using lambda functions.

Define: **demonstrate_player_transformations(players)**

This function should:

- Accept a list of player dictionaries as parameter
- Validate input type and handle empty lists gracefully
- Filter out invalid player dictionaries (missing required keys)
- Use `map()` with lambda functions to transform player data at least 3 different ways:
    - Calculate effective health (health + level*10) using `map(lambda p: {"name": p["name"], "effective_health": p["health"] + p["level"]*10}, players)`
    - Calculate mana regeneration per turn using level multiplier
    - Calculate normalized score (score divided by level)
    - Calculate power index using custom formula combining health, mana, and level
- Convert all map results to lists for display
- Print results with clear labels and formatting
- Handle players missing required attributes by skipping them
- Display each transformation result with player names and calculated values

6. Write a Python function to demonstrate entity filtering using lambda functions.

Define: **demonstrate_entity_filtering(entities, player_position=(100, 100))**

This function should:

- Accept a list of entity dictionaries and optional player position tuple
- Validate input types and handle empty lists gracefully
- Filter out invalid entity dictionaries (missing required keys)
- Use `filter()` with lambda functions for at least 3 different filtering operations:
    - Filter active enemies: `filter(lambda e: e["type"] == "enemy" and e["active"], entities)`
    - Filter collectible items that are active
    - Filter entities within specified distance from player using distance formula
    - Filter entities in specific quadrants based on position coordinates
- Convert all filter results to lists for display
- Calculate distances using formula: `((e["position_x"] - player_x)**2 + (e["position_y"] - player_y)**2)**0.5`
- Print results with entity IDs, types, and positions
- Handle entities missing required attributes by skipping them
- Display count and details of filtered entities for each operation

7. Write a Python function to demonstrate item sorting using lambda functions.

Define: **demonstrate_item_sorting(inventory)**

This function should:

- Accept a list of item dictionaries as parameter
- Validate input type and handle empty lists gracefully
- Filter out invalid item dictionaries (missing required keys)
- Use `sorted()` with lambda functions for at least 3 different sorting operations:
    - Sort by value ascending: `sorted(inventory, key=lambda item: item["value"])`
    - Sort by rarity using custom order: define rarity_order dict and use `sorted(inventory, key=lambda item: rarity_order.get(item["rarity"], 0))`
    - Sort by type then value descending: `sorted(inventory, key=lambda item: (item["type"], -item["value"]))`

    - Filter equipped items and sort by value
- Create rarity ordering: `{"common": 0, "uncommon": 1, "rare": 2, "epic": 3, "legendary": 4}`
- Print results showing item names, types, values, and rarity levels
- Handle items missing required attributes by skipping them
- Display sorted results with clear section headers for each sorting method

8. Write a Python function to demonstrate game calculations using lambda functions.

Define: **demonstrate_game_calculations(coordinates, player_data)**

This function should:

- Accept lists of coordinates and player data as parameters
- Validate input types and handle empty lists gracefully
- Check for minimum data requirements (at least 2 coordinates, 1 player)
- Use lambda functions for at least 3 different game calculations:
    - Calculate distances between consecutive coordinates using `map(lambda i: distance_formula, range(1, len(coordinates)))`
    - Create damage calculation lambda: `lambda attacker, defender: formula_based_on_level_and_stats`
    - Create movement speed lambda: `lambda player: base_speed + level_bonus - health_penalty`
- Calculate total path length by summing distances
- Use distance formula: `((x2-x1)**2 + (y2-y1)**2)**0.5`
- Print calculated distances, total path length, damage between players, and movement speeds
- Handle invalid data by skipping calculations and showing appropriate messages
- Display results with clear formatting and units

5.3 Advanced Demonstration Functions

9. Write a Python function to demonstrate game ability system using lambda functions.

Define: **demonstrate_ability_system()**

This function should:

- Define a dictionary of game abilities using lambda functions
- Create at least 4 different abilities with level scaling:
  - Fireball: `lambda level: {"damage": 20 + level * 5, "mana_cost": 10 + level * 2}`
  - Heal: `lambda level: {"healing": 15 + level * 5, "mana_cost": 15 + level * 3}`
  - Shield: `lambda level: {"defense": 10 + level * 3, "duration": 2 + level // 2}`
  - Lightning: `lambda level: {"damage": 15 + level * 7, "mana_cost": 20 + level * 4}`
- Show ability stats at different levels (1, 3, 5)
- Create ability usage checker: `lambda player, ability, level: player["mana"] >= ability(level)["mana_cost"]`
- Test ability usage with sample player data
- Print ability statistics and usage possibilities
- Format output with clear ability names and stat breakdowns

10. Write a Python function to demonstrate a combat system using lambda functions.

Define: **demonstrate_combat_system(players, entities)**

This function should:

- Accept lists of players and entities as parameters
- Validate input types and handle empty lists or invalid data
- Filter valid players and entities with required attributes
- Use lambda functions for combat calculations:
  - Find enemies in attack range: `filter(lambda e: distance_to_player <= attack_range and e["type"] == "enemy", entities)`
  - Calculate hit chance: `lambda distance: max(0, min(100, 100 - distance * 0.5))`
  - Calculate damage: `lambda player, distance: max(1, player_stats * distance_modifier)`
  - Calculate XP reward: `lambda distance: base_xp - distance_penalty`

- Simulate combat with first player at position (100, 100)
- Use attack range of 150 units
- Print enemies in range, combat calculations for each enemy
- Display hit chances, potential damage, and XP rewards

11. Write a Python function to demonstrate a level system using lambda functions.

Define: **demonstrate_level_system(players)**

This function should:

- Accept a list of player dictionaries as parameter
- Validate input type and handle empty lists or invalid data
- Filter valid players with required attributes
- Use lambda functions for level progression:
  - Calculate XP required: `lambda level: 100 * (level ** 1.5)`
  - Calculate stats at level: `lambda base_stats, level: {"health": base_health + level * 10, "mana": base_mana + level * 5}`
- Show XP requirements for levels 1-10
- Calculate base stats by reverse engineering current player stats
- Show stat progression for each player at next 3 levels
- Print XP requirements table and player progression forecasts

5.4 Main Program Function

12. Write a Python function to demonstrate the complete game development utility system.

Define: **main()**

This function should:

- Print the system header: "===== GAME DEVELOPMENT UTILITY SYSTEM ====="
- Call all data preparation functions to get game data
- Execute all demonstration functions in order:
  - demonstrate_player_transformations(players)
  - demonstrate_entity_filtering(entities)

    - demonstrate_item_sorting(inventory)
    - demonstrate_game_calculations(coordinates, players)
    - demonstrate_ability_system()
    - demonstrate_combat_system(players, entities)
    - demonstrate_level_system(players)
- Print final completion message: "===== DEMONSTRATION COMPLETED ====="
- Ensure all functions execute without errors

6. EXECUTION STEPS TO FOLLOW:

1. Run the program

2. Observe player data transformations with lambda functions

3. See game entity filtering with lambda functions

4. View inventory sorting with lambda functions

5. Observe game mechanic calculations with lambda functions

6. See ability system demonstrations

7. View combat system calculations

8. Observe level system progression calculations

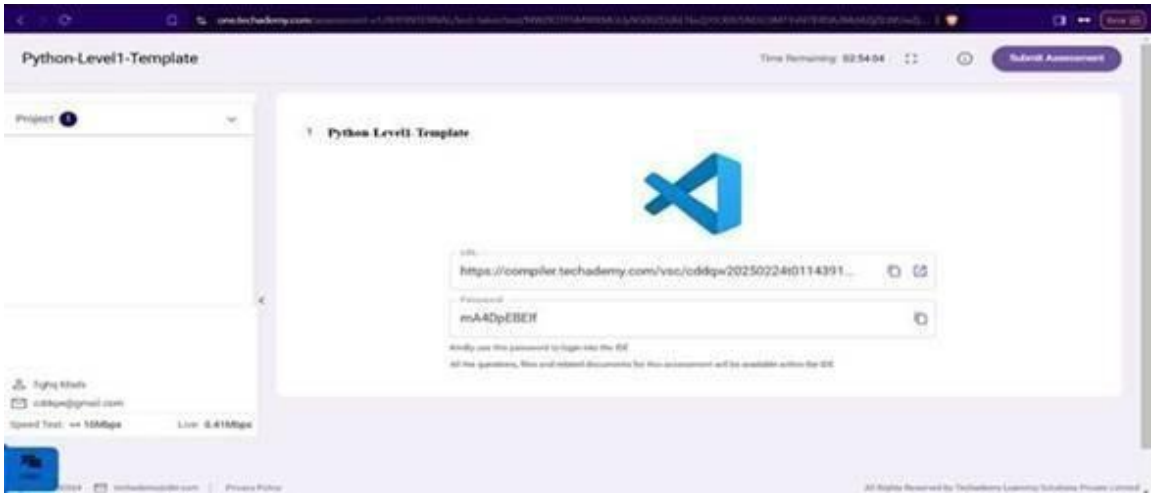
Execution Steps to Follow:

- All actions like build, compile, running application, running test cases will be through Command Terminal.
- To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal
- This editor Auto Saves the code
- If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page)

- These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.
- To launch application: **python3 filename.py**
- To run Test cases: **python3 -m unittest**

Screenshot to run the program

- To run the application **python3 filename.py**
- To run the testcase **python3 -m unittest**



- Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.