# Development Log for Software Design: "Title of project”

# Project Title: Student Management System Date Started: January 1, 2025 Development Period: X Weeks.

# Author:

Table of Contents

[Development Log for Software Design: "Title of project” 1](#_Toc197280511)

[Project Title: Student Management System Date Started: January 1, 2025 Development Period: X Weeks. 1](#_Toc197280512)

[Author: Name Here. 1](#_Toc197280513)

[Structure Chart 2](#_Toc197280514)

[Program Design (Pseudocode/Flowcharts) 2](#_Toc197280515)

[Function Description 8](#_Toc197280516)

[Testing and Version Control 11](#_Toc197280517)

[Evidence of Integration Testing 13](#_Toc197280518)

[Evidence of Version Control 14](#_Toc197280519)

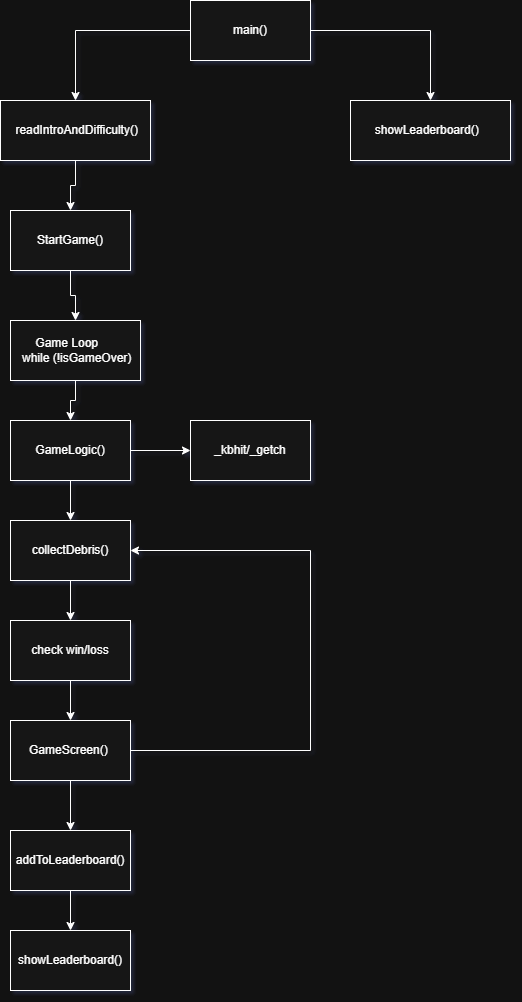
## Structure Chart

The structure chart indicates a neat hierarchical design with **main()** being the control module, and support functions used for the initialization, input/output, game logic processing, and file handling. This modular design enhances the understanding, testing, and extensibility of the program, thus adhering to good software engineering practice.



## Program Design (Pseudocode/Flowcharts)

**Function Interaction Diagram (Flowchart Representation)**  
Below is a flow diagram for the control and interaction among major functions of the space survival game, showing how the game initializes, runs, and terminates.

****

**Explanation of the Flow:**

* **main():** Takes care of the main menu and user selections and calls **readIntroAndDifficulty()** to select difficulty.
* **StartGame(): Initializes player and asteroid positions.**
* Game Loop (while (!isGameOver)): Repeats:
  + **GameScreen()** is called to render the current state of the game.
    - **GameLogic()** is executed to capture player inputs and execute game mechanics, including:
      * Keyboard input through **\_kbhit/\_getch**
      * Player movement, asteroid movement
      * Calls **collectDebris()** when junk is collected
      * Ends loop if win/loss condition is met**.**
* **When game ends:**
  + Leaderboard is updated through **addToLeaderboard().**
  + Shows leaderboard through **showLeaderboard().**

**Pseudocode (Detailed Function Logic)**

This section describes the internal logic behind the main control system of the game and supporting functions in the form of structured pseudocode, thereby fostering modular and readable game design.

**main.c – Game Control Flow:**

START main

WHILE true

CLEAR screen

DISPLAY menu:

1. Start Game

2. Show Leaderboard

3. Exit

READ user input into menuSelection

IF menuSelection == 1 THEN

CALL readIntroAndDifficulty()

IF valid difficulty selected THEN

CALL StartGame()

WHILE game is not over

CALL GameScreen()

CALL GameLogic(player)

WAIT movementCooldown milliseconds

END WHILE

CALL GameScreen()

IF win condition met THEN

DISPLAY "You win!"

ELSE

DISPLAY "Game Over"

CALL addToLeaderboard(score, fuel, health)

CALL showLeaderboard()

RESET player and game state

WAIT for key press

END IF

ELSE IF menuSelection == 2 THEN

CALL showLeaderboard()

WAIT for key press

ELSE IF menuSelection == 3 THEN

EXIT program

END IF

END WHILE

END main

**Function Definitions – main.c**

**readIntroAndDifficulty():**

OPEN settings.txt

DISPLAY file contents line-by-line

REPEAT until valid input

PROMPT user to select difficulty (1–3)

IF valid THEN

SET initial fuel, asteroid count, and cooldown

ELSE

DISPLAY "Invalid input"

RETURN to menu

**StartGame():**

SET player position to center of grid

SET health = 100, score = 0, fuel based on difficulty

RANDOMIZE debris position

FOR each asteroid

RANDOMIZE asteroid position

ENSURE not too close to player

**GameScreen():**

**CLEAR console**

**FOR each row and column in grid**

**IF player is at this cell → print "A"**

**ELSE IF debris is at this cell → print "#"**

**ELSE IF asteroid is at this cell → print "@"**

**ELSE → print "."**

DISPLAY player stats (Fuel, Health, Score)

DISPLAY controls reminder

**GameLogic(Player *p):***

IF key pressed THEN

HANDLE input (w/a/s/d/h/x)

DECREASE fuel

FOR each asteroid

MOVE randomly (up/down/left/right)

IF asteroid collides with player

DECREASE player health

IF health ≤ 0 THEN set game over

IF player position matches debris

CALL collectDebris(p)

RANDOMIZE new debris location

CHECK if fuel or health == 0 → set game over

CHECK if score >= 5 → set win

**\**collectDebris(Player p):***

INCREASE health up to 100 max

INCREASE fuel up to 100 max

INCREMENT score

**utilis.c – File I/O Functions**

**addToLeaderboard(score, fuel, health):**

OPEN leaderboard.txt in append mode

WRITE score, fuel, and health as a new entry

CLOSE file

**showLeaderboard():**

OPEN leaderboard.txt in read mode

IF file exists

PRINT each line

ELSE

DISPLAY "Leaderboard is empty"

CLOSE file

## Function Description

The program has modular components (functions) to show how responsibilities are allocated and how functions relate to each other.

Each function in the system will now be described below:

**readIntroAndDifficulty():**

* **Description:** The function reads the game intro text and difficulty settings from a file (i.e. settings.txt) and prompts the user to choose a difficulty. The game parameters (fuel, count of asteroids, and movement speed) are set up on the basis of the chosen difficulty.
* **Input Parameters:**
  + None (reads from settings.txt and user input).
* **Returns:**
  + None. It modifies global variables according to user inputs, gameDifficulty, movementCooldown, totalAsteroids, and player.fuel.

**StartGame():**

* **Description:** Initializes the game state, with the player's position, health, fuel, and score all set randomly, thereby placing debris and asteroids on the game grid.
* **Input Parameters:**
  + None.
* **Returns:**
  + None. It sets up the initial game state in global variables (player.x, player.y, player.health, player.fuel, player.score, debrisPosX, debrisPosY, etc.).

**GameScreen():**

* **Description:** It clears the screen and visually assembles the current game grid that indicates the player as "A", debris as "#", asteroids as "@", and free space as ".". There are also parameters displayed at the bottom, which are the fuel, health, and score.
* **Input Parameters:**
  + None (reads from the global variables player, debrisPosX, debrisPosY, asteroidPositionsX, asteroidPositionsY).
* **Returns:**
  + None. It only renders the game state to the console**.**

**GameLogic(Player \*p):**

* **Description:** Manages the main game logic. Takes user input (movement commands and actions), updates player's position, decreases fuel, moves the asteroids, checks for collisions, then evaluates game-over or win conditions**.**
* **Input Parameters:**
  + Player \*p: Pointer to the player struct. It contains the player's current position, health, fuel, and score**.**
* **Returns:**
  + **None**. It modifies the player struct and sets global flags (isGameOver) based on game conditions.

**collectDebris(Player \*p):**

* **Description:** Called when the player collides with debris, it increases the player's health (up to a maximum restore of 100), fills fuel (up to a maximum restore of 100), and increments the player's score.
* **Input Parameters:**
  + Player \*p: Pointer to the player struct, containing the player’s current health, fuel, and score.
* **Returns:**
  + **None.** It only modifies the player struct by increasing health, fuel, and score.

**addToLeaderboard(int score, int fuel, int health):**

* **Description:** This function saves the score, fuel, and health of the player in a file called leaderboard.txt. It is called whenever the game comes to an end and saves the player's performance against which the player can be measured in the future.
* **Input Parameters:**
  + **score**: Integer variable, representing the player's final score**.**
  + **fuel:** Integer variable, representing the player's remaining fuel.
  + **health:** Integer variable, representing the player's remaining health**.**
* **Returns:**
  + **None**

**showLeaderboard():**

* **Description:** Displays the contents of the 'leaderboard.txt' file depicting high scores of the previous players and notifies the user if it displays an empty input leaderboard.
* **Input Parameters:**
  + **None.**
* **Returns:**
  + None. It simply prints the leaderboard contents to the console, or a message if the file is empty.

**main():**

* **Description:** This is the main entry point of the program. It handles the game menu, user input, invokes functions like StartGame(), and displays the leaderboard. It also manages game flow (starting the game, checking for game over conditions, displaying results).
* **Input Parameters:**
  + None.
* **Returns:**
  + None. It is the top-level control function that drives the program by calling other functions based on user input.

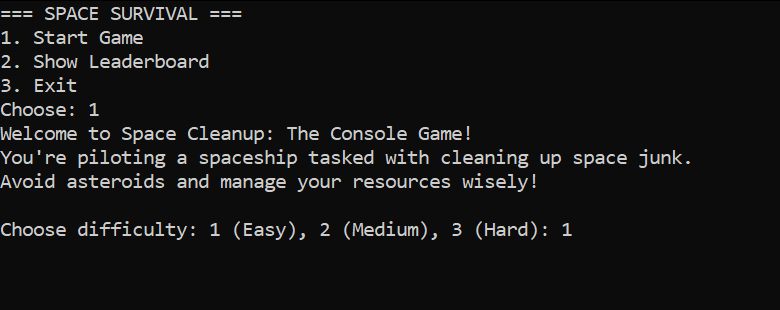
## Testing and Version Control

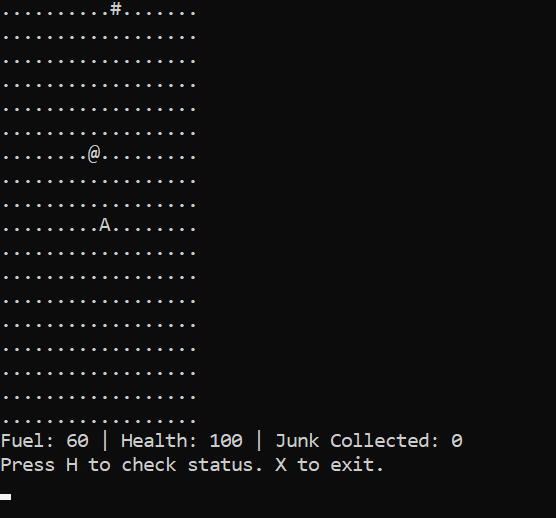
**Function: readIntroAndDifficulty()**

* Purpose: Sets game difficulty and adjusts fuel/asteroid values.

Test Case 1: Valid Input

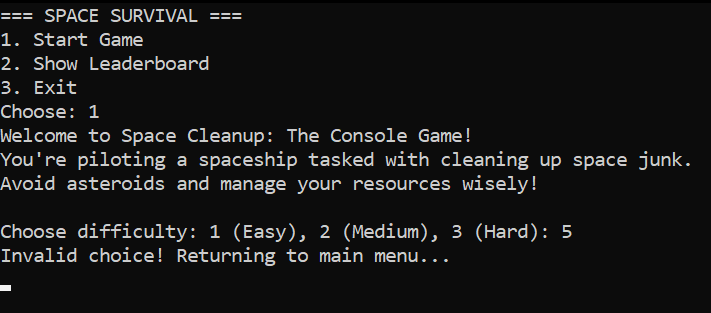
* Input: 1 (Easy)
* Expected Output: Fuel = 60, Asteroid Count = 1
* Result: Passed





Test Case 2: Invalid Input

* Input: 5
* Expected Output: Prompts the user again for input and returns to menu
* Result: Passed

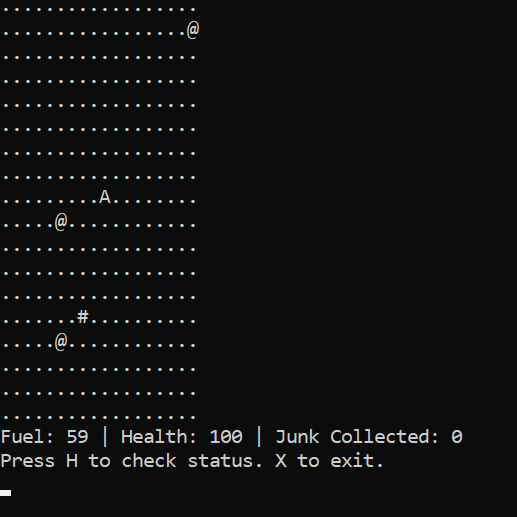


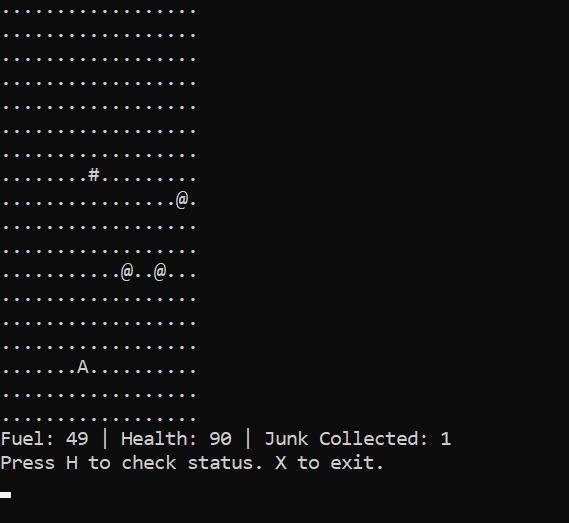
**Function: GameLogic(Player \*p)**

* Purpose: Handles movement, fuel/health decrease, collisions, and win/loss detection.

Test Case 1: Player Collision with Asteroid

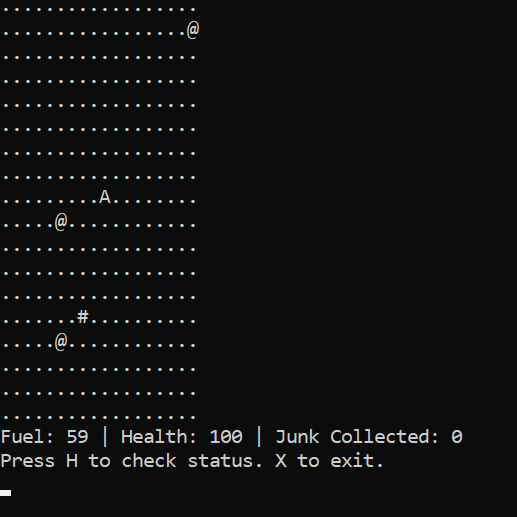
* Input: Player at (5,5), asteroid moves to (5,5)
* Expected Output: Health decreases, possible game over
* Result: Passed





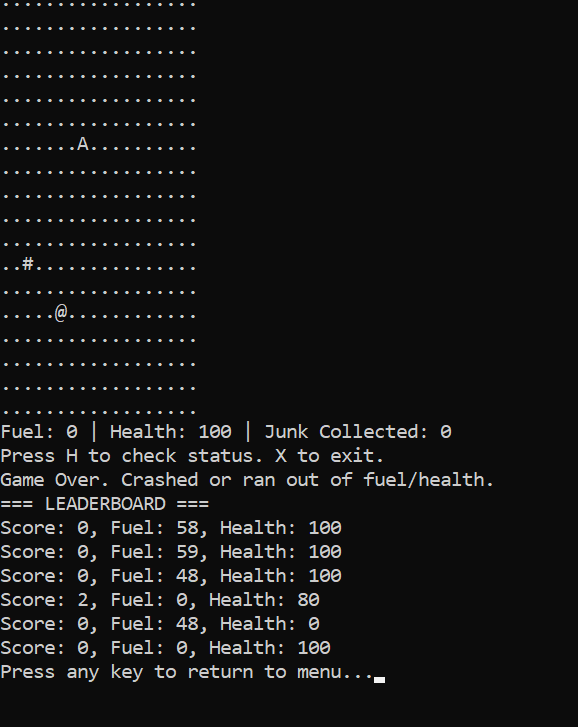
Test Case 2: Movement Input 'w'

* Input: 'w' key pressed
* Expected Output: Player moves up, fuel decreases
* Result: Passed



Test Case 3: Fuel Depletion

* Input: Fuel = 0
* Expected Output: Game ends
* Result: Passed

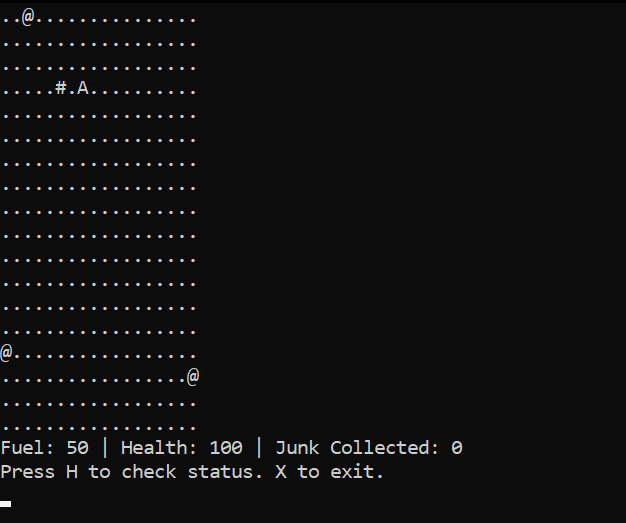


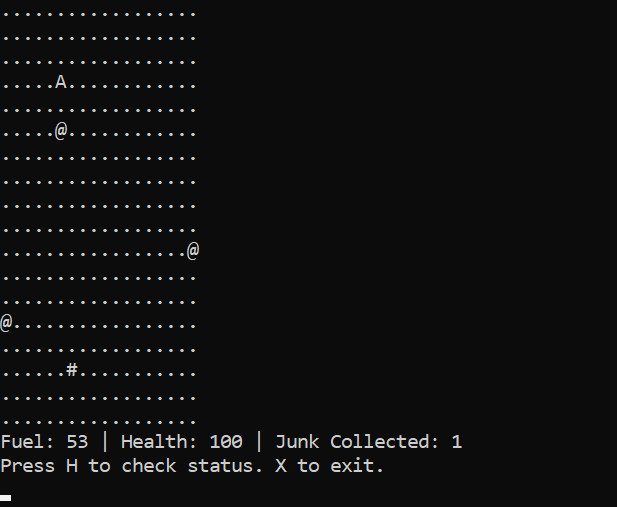
**Function: collectDebris(Player \*p)**

* Purpose: Increases score, health, and fuel when debris is collected.

Test Case 1: Collect Debris with <100 Health and Fuel

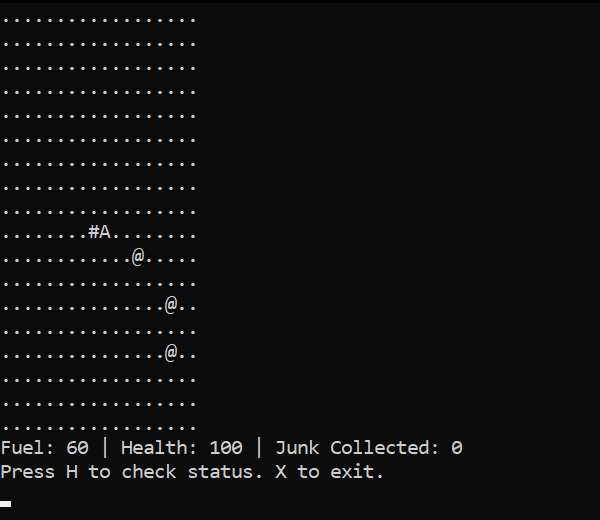
* Input: Health =100, Fuel = 60
* Expected Output: Health increases (max 100), Fuel increases
* Result: Passed

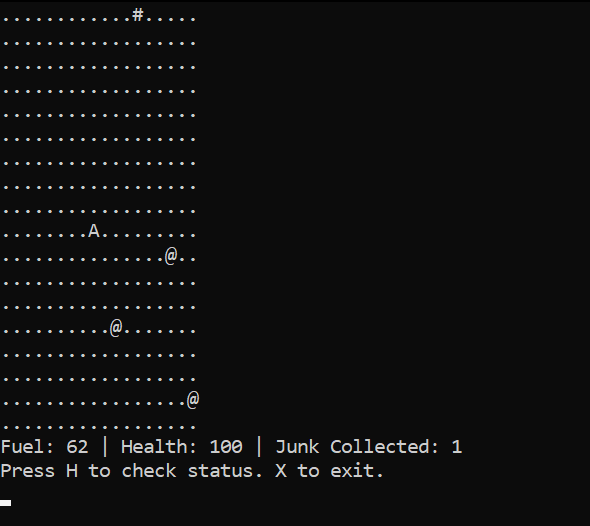




Test Case 2: Health/Fuel Already at 60

* Input: Health = 100, Fuel = 60
* Expected Output: No increase beyond 60
* Result: Failed



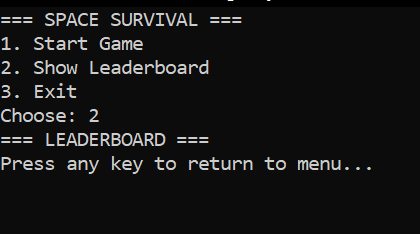


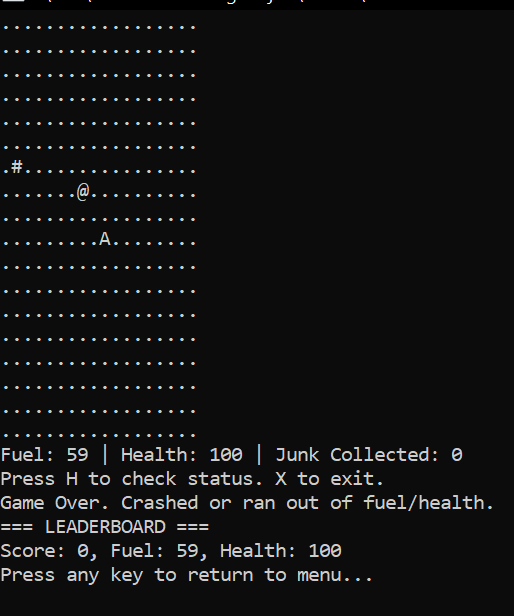
**Function: addToLeaderboard()**

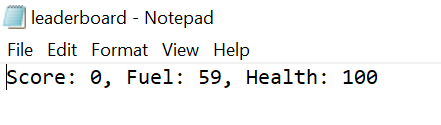
Purpose: Appends player score, fuel, and health to leaderboard.txt.

Test Case 1: Normal Input

* Input: Score = 0, Fuel = 60, Health = 100
* Expected Output: Line added to file
* Result: Passed







**Function: StartGame()**

Purpose: Initializes player, debris, and asteroid positions.

Test Case 1: Player Spawn Position

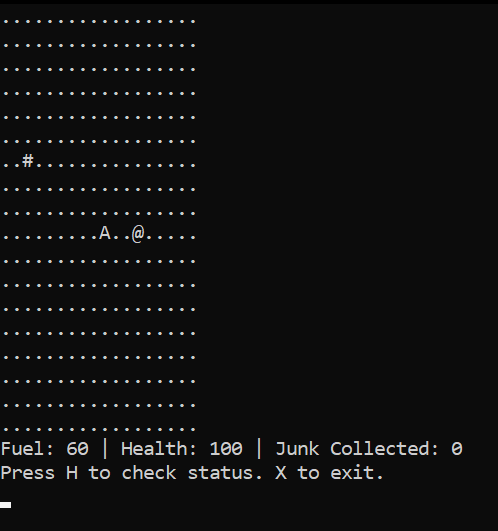
* Input: Game start
* Expected Output: Player spawns at center of grid
* Result: Passed



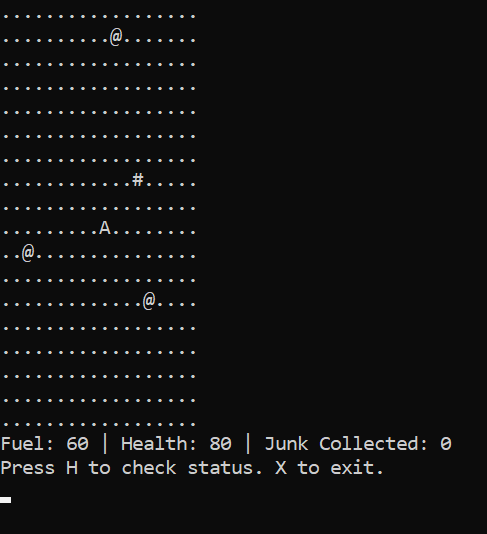
Test Case 2: Maximum Debris Spawn Limit

* Input: Game initializes with maximum allowed debris count for each level (e.g., 1 for Easy, 2 for Medium and 5 for Hard).
* Expected Output: All debris spawn at non-overlapping, valid positions on the map.
* Result: Passed

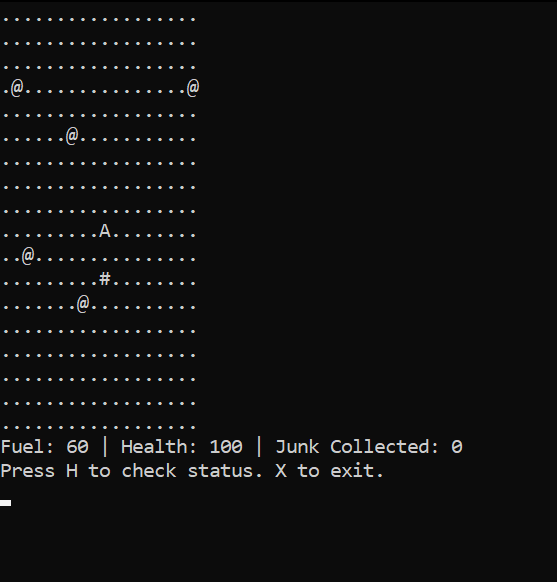
Max Asteroid Count for Easy:



Max Asteroid Count for Medium:



Max Asteroid Count for Hard:

****

**Unit Test Results Summary:**

**Overall Outcome:**  
All the functions are tested separately and passed under expected conditions. Each function behaved correctly with both normal and edge inputs.

**Edge Case Testing:**  
Invalid difficulty selection, depletion of fuel at 0, an asteroid colliding at the same coordinates, health/fuel limited to 100 were the edge cases tested, and successful testing was achieved.

**Intended Outstanding Issues:**  
There exist no significant functional bugs. However, user input emergency handling measures could be improved in such a way that accidental infinite loops are not encountered due to unintentional user inputs or system state changes resulting in application crashes.

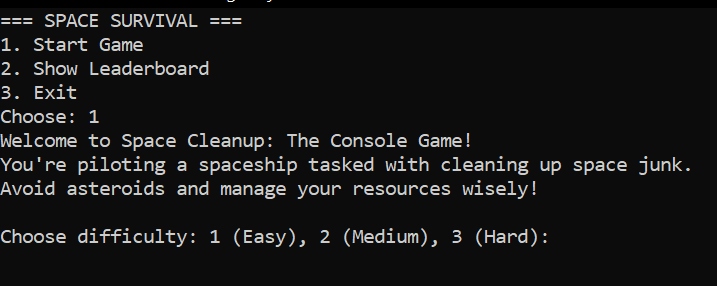
## Evidence of Integration Testing

**These blocks/functions were tested together:**

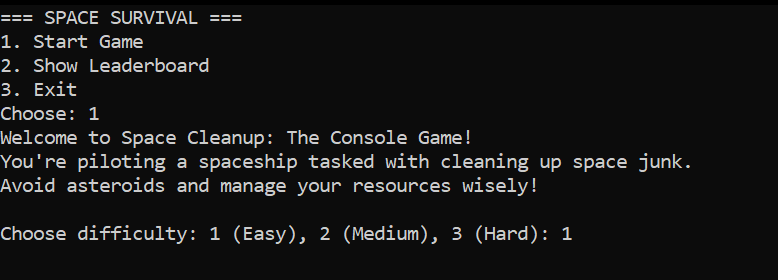
* readIntroAndDifficulty() and StartGame()
* GameLogic() and GameScreen()
* GameLogic() and collectDebris()
* main() and addToLeaderboard()

**Test Steps:**

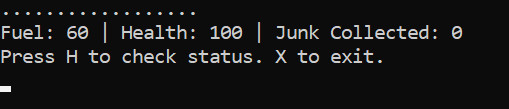
1. Run the application, and select Start Game (Choose input option 1) which triggers the main menu and readIntroandDifficulty() function.



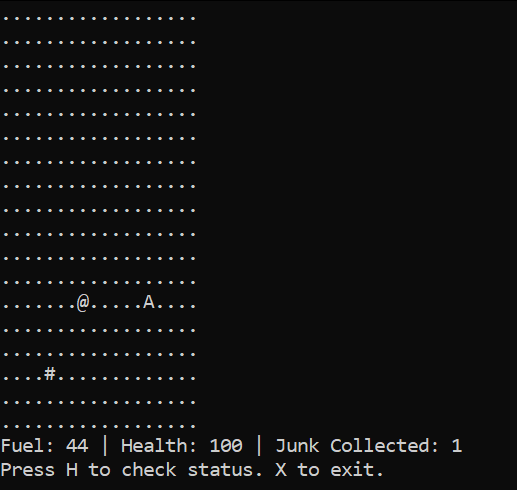
1. Select the difficulty level (i.e., 1 for Easy mode), which will trigger the GameLogic() function to display the game grid



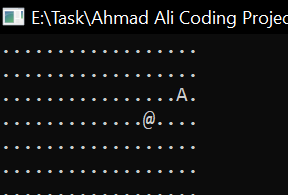
1. Check that the values of fuel and asteroid are being properly initialized via StartGame().

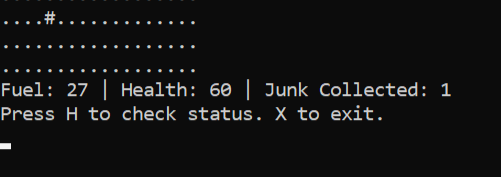


1. Move player and collect debris; this will cause the interaction between GameLogic() and collectDebris() to be invoked, incrementing Junk Counter as the player plays the game.



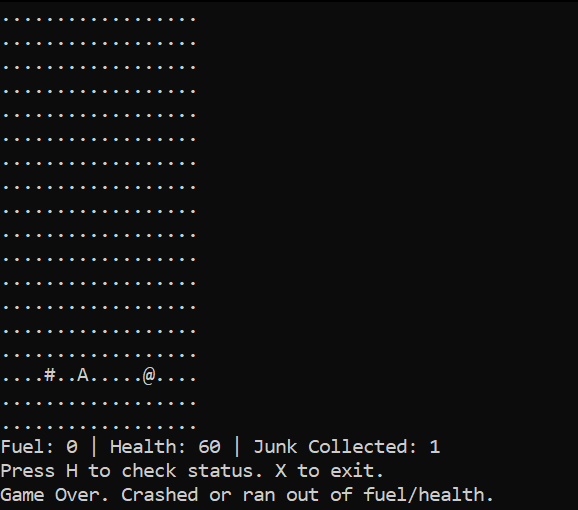
1. Tracking asteroid movement down to collision response in the game loop with the GameLogic() + GameScreen().





1. Win condition achieved or health/fuel out, finish game.

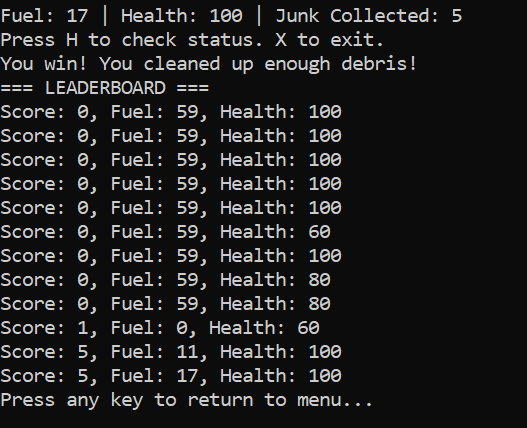
Game Over, ran out of Fuel:

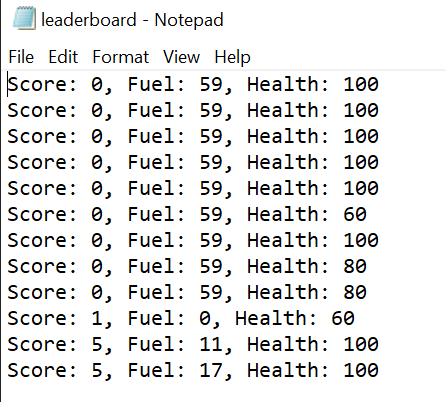


Player wins, collecting enough Junk:



1. The score is confirmed to be saved in a file via addToLeaderboard() and shown after the game ends via showLeaderboard()





**Expected Outcome:**

* Program should smoothly transition from the menu to difficulty selection, to gameplay, and finally to end-game screens.
* readIntroAndDifficulty() correctly initializes difficulty settings used by StartGame().
* GameLogic() correctly calls collectDebris() when the player intersects the debris.
* GameScreen() refreshes on the basis of real-time logic from GameLogic().
* addToLeaderboard() will mark the final game state.

**Result:**

**Passed** — all interactions between functions were as expected with no interrupting logical breakpoint or crashing.

**Integration Test Results Summary:**

The system correctly integrates:

* readIntroAndDifficulty() with StartGame() to configure game based on player selection.
* GameLogic() and GameScreen() for real-time visual updates based on logic.
* GameLogic() and collectDebris() for updating health/fuel/score mid-game.
* main() with addToLeaderboard() for end-game record keeping.

## Evidence of Version Control

The project development did not use Git or Github, and yet version control was manually administered by saving the different versions in different files. The project underwent three major versions addressing significantly enough problem and improvement with functionality. Below is the outline of the development process with indications of versioning:

**Version 1.0 – Initial Implementation**

* Features: Basic game structure inspired by the ‘snake game’ with menu, player movement, asteroids, and debris collection.
* Issues:
  + Difficulty selection did not work properly—asteroid count was not changing per level.
  + Leaderboard was not updating on game over or win conditions.

Filename: main\_v1.c

**Version 2.0 – Menu and Settings Fixes**

* Changes:
  + Some attempt was made to fix the logic for the difficulty selection.
  + Implemented reading from settings.txt to show the intro.
* Issues:
  + settings.txt read before the menu, breaking the expected flow.
  + Writing to the leaderboard still did not work..

Filename: main\_v2.c

**Version 3.0 – Final Functional Version**

* Changes:
  + Incorporated pointer-based Player Struct for dynamic attribute management(fuel, health, score).
  + Revamped menu flow; intro and difficulty prompts only appear after menu selection.
  + Fixed leaderboard updates.
  + Added header files to furnish code organization.
* Status: Fully functional, with difficulty settings, player win/loss condition handling, and storing leaderboard.

Filename: main\_final.c

**Version Control Summary:**

* The versioning was done manually with separate source code files (main\_v1.c, main\_v2.c, main\_final.c).
* Every version was reviewed and tested to ensure that suitable improvements were made before the next version commenced.
* All bug fixes related to difficulty scaling, persistence of the leaderboard, and menu logic were enacted in the final version.

Git usage would have resulted in the establishment of branches for:

* + feature/difficulty-selection
  + bugfix/leaderboard-write
  + refactor/pointer-struct-logic

## References

**Software Design & Development**

Ruas, T. & Grosky, W. (2020) Software engineering: a practitioner's approach, 9th edn. New York: McGraw-Hill.

Sommerville, I. (2016) Software engineering, 10th edn. Harlow: Pearson. Available at:

<https://dn790001.ca.archive.org/0/items/bme-vik-konyvek/Software%20Engineering%20-%20Ian%20Sommerville.pdf>

**Software Testing & Quality Assurance**

Kaner, C., Falk, J. & Nguyen, H.Q. (1999) Testing computer software, 2nd edn. New York: Wiley. Available at: <https://books.google.com.pk/books?id=Q-hTDwAAQBAJ&printsec=frontcover>

Myers, G.J., Sandler, C. & Badgett, T. (2012) The art of software testing, 3rd edn. Hoboken: Wiley. Available at: <https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf>

**Project Documentation & Best Practices**

IEEE (2009) \*IEEE Std 1016-2009: IEEE standard for information technology—system design—software design descriptions\*. New York: IEEE. Available at: <https://cengproject.cankaya.edu.tr/wp-content/uploads/sites/10/2017/12/SDD-ieee-1016-2009.pdf>

McConnell, S. (2004) Code complete, 2nd edn. Redmond: Microsoft Press. Available at: <https://people.engr.tamu.edu/slupoli/notes/ProgrammingStudio/supplements/Code%20Complete%202nd.pdf>