Heriot-Watt University Dubai

*Praxis Programming B37VB, Year 1*

Dungeon.txt - C Game Final Report

|  |  |
| --- | --- |
| ***Student’s name*** | Aerold Parcon |
| ***Student ID*** | H00448855 |
| ***Professor’s name*** | Dr Girish Balasubramanian |

# Abstract

DUNGEON.txt, a terminal-based dungeon crawler, was implemented as part of a project for Praxis Programming to learn how to implement software using the C programming language. Key C programming principles such as functions, text file parsing, low-level memory management, input and output, structures and pointers were utilised, offering a modular and scalable engine for the game.

# Introduction

This report highlights the development and implementation of Dungeon.txt, a terminal-based dungeon crawler game in C. Drawing inspiration from Dungeons & Dragons' dungeon crawling and Fire Emblem's turn-based tactical combat, this game incorporates grid-based movement in a 2D tile map, a modular .txt file level design system, a text-based entity defining system. A text file-based dialogue system and a strategic battle system utilizing an initiative-based turn order implemented with a circular linked list. Enemy behaviour and pathfinding using a breadth first search algorithm and states. An ASCII rendered tile map with fog of war implemented through a recursive shadow casting algorithm.

**Aims**

Gain hands on experience in using the C programming language by implementing a fully functioning game.

**Objectives**

* Implement data structures for different objects in the game, mimicking OOP
* Manage memory for dynamically allocated structures, arrays and data structures
* Parse text files for rooms, entities, items and dialogue
* Print a text-based ASCII GUI on the terminal for the game
* Handle user input and control a player which can affect the world
* Implement monster AI using a pathfinding algorithm
* Create a turn based grid combat system.

# Game loop

Game logic is usually structured around a main game loop. According to Game Programming Patterns, a book by Robert Nystrom (2014), a game loop runs continuously during gameplay. Each turn of the loop, it processes user input without blocking, updates the game state, and renders the game. It tracks the passage of time to control the rate of gameplay.

Dungeon.txt runs using a game loop. Its main game loop can be summarised by the flowchart in figure 1.

***Figure 1:*** *Flowchart demonstrating the main loop.*

# Project structure

To ensure the modularity, readability and encapsulation of different functionalities in the game, the main source code has been split up into different modules under different sub directories. Header files for each C source file were used so different modules can include each other. The file directory used in the game is available on Appendix A.

Github version control was used. Source code and the project’s commit history is available on the Github page <https://github.com/aeroldd/dungeon.txt>.

Due to multiple source files being used, the project had to be compiled in a specific way, with the different subdirectories. To simplify the compilation process, CMake was used. CMake is a widely used software which can be used to build C source code with different options for different platforms and multiple source files (Kitware, 2025). The CMake file is available on the Github page, as well as instructions for using CMake to compile and build the game.

# Mechanics

## Room and map system

The main feature of dungeon crawlers is map exploration through either a randomly generated or a curated dungeon. A modular and scalable level design was implemented by parsing text files containing room information during runtime.

RoomGrid \*createRoomGrid(char\* roomName)

The createRoomGrid() function parses a file from assets/rooms with the name roomName and stores data such as the room layout and entities in the room in a RoomGrid struct. An example of a room level text file is attached in appendix A. The room tile grid is stored dynamically in a 2-dimensional array of Tile structs. Doors and entities are stored in a dynamically allocated 1-dimensional array.

These rooms are stored in a higher-level Map struct which stores and handles running rooms and transitioning between rooms with doors. Doors store the targeted room’s id and player’s new position, and when interacted with, will load the room as the active room and place the player there.

## Running entities

Similarly to rooms, entities are loaded from text files.

Entity \*createEntityFromFile(char\* entityName, Position mapPos, Position gridPos, int currentHealth)

The createEntityFromFile() function parses a text file from the assets/entity directory and stores data in an Entity struct. The OOP concept of polymorphism and inheritance are emulated in C using the type attribute in the Entity struct which dictates what function to use when running the entity from the runEntity() function. An example of an entity file is available on Appendix B.

### Initiative and turn order

Running entities in this game are implemented using an initiative list (Wizards of the Coast, 2014, p. 189). In the game, when a room is loaded, all entities roll a number between 1 and 20, with their initiative modifier added. The ids of all of these entities are added to a circular linked list - called InitiativeList in game - which cycles through the entities and runs them using the runEntity() function.

## Running the player

Player actions and inputs are handled using the getInput() function, written to use conio.h’s getch() function for Windows systems, or termios.h’s keyboard handling functions to ensure cross platform support. The player moves around in a 2D tile-based array system using WASD keys with collision detection.

The action menu function displays dynamic menu options calls different functions according to the user input which change the games state, such as combat, movement and inventory management.

## Combat

When the room isn’t cleared, player movement is restricted by the speed stat. Players have the option to attack adjacent monsters once during their turn. When multiple targets are available, the player can choose the target using the spacebar. Attack calculations are done using a 20-sided die (using rand()) and adding the attacker’s attack modifier. The attacker hits if the value is greater than the target’s armour stat. Once all movement and attacks have been done, the player’s turn is over and the next entity’s turn is cycle through.

## Monster behaviour and AI

During monster turns, if the monster can detect a player in its detection radius, it uses a breadth first search algorithm to determine the shortest path to the player. It moves to the player to its maximum speed. The monster attacks the player once it reaches the player. Monsters which aren’t in the player’s detection range roam around randomly.

## Items, equipment, consumables and polymorphism

Different items and their properties are defined in text files.

Item \*createItemFromFile(char\* itemName, int itemType)

The createItemFromFile() function parses for a file named itemName in assets/items with the according type and creates an Item struct with its attributes. Polymorphism is used to define different types of items with different attributes and behaviours and inheriting shared functions. This is done with structs – a base Item struct stores the shared attributes, and these shared attributes are accessed by the equipment with its base attribute which stores a pointer to its base item. Pointer type casting is also used to retrieve parent attributes.

These items are dynamically stored in an Inventory struct which contains a list of objects and has a text-based GUI which shows selected items, and inventory options. Items can be equipped, dropped or used depending on the type of object which accurately updates the game state. Item dropping ensures proper memory management with freeing the memory.

An example of an item file is available on the appendix.

## Text based dialogue system

A modular dialogue system with dynamic keyword formatting and replacing was implemented using text file parsing.

void playDialogue(const char\* filename, char \*\*keywords, char \*\*replacements, int keywordCount)

The playDialogue() function parses through a text file in the assets/dialogue subdirectory and outputs the lines using the fancyPrint() function which prints characters one by one. Keywords in the dialogue text file denoted with {curly braces} are replaced with replacements which are defined in arrays. Dialogue is played when entering rooms and interacting with the game. An example of a dialogue text file is available on Appendix D.

## Game rendering

Game is run on the terminal. Graphics are implemented using ASCII characters which are print using the stdio.h printf() and the fancyPrint() function. All game objects rendered to the screen have a symbol attribute which were used in rendering. The process below demonstrates the rendering order when the displayRoomWithCamera() function is invoked to display the map centred around a targeted entity.

1. Calculate bounds of the tile array using the camera size.
2. Calculate tile visibility - uses a shadowcasting algorithm to determine what the target entity can see. The shadowcasting algorithm used was adapted from Albert Ford’s explanation of FOV rendering for roguelikes (Ford, 2018)
3. Render room tiles: Iterate through each tile in the room’s tiles 2D array and print its symbol attribute according to its visibility.
4. Render entities: Iterate through the room’s entity array and print its symbol attribute if the tile below it is visible.

The screen is updated using the system(“cls”) function for Windows and system(“clear”) for MacOS and Linux which clears the screen so the updated screen can be printed on.

Alongside the map, the player’s stats including health, armour, distance left was displayed and updated accordingly.

# Evaluation

A modular engine to make a user designed dungeon was implemented and it provides customisability and scalability to level design. JSON files could have been used for better data organisation inside asset files.

Current text file parsing functions lack robust error handling. For example, slight file formatting errors can cause improper memory allocation and program crashes. Entity and door placing from room text files lack proper out of bounds and valid placing protection which could crash the program when entities try moving outside of boundaries. Proper boundary checking and error handling should be implemented for future versions of this game.

ASCII text rendering system using the standard input output library works as intended and gives a retro look similar to Rogue, the original text-based dungeon crawler (Toy, Wichman, & Arnold, 1980). However, due to constant console clearing, the screen flickers during frame updates. An alternative renderer using ncurses - a programming library to write text-based user interfaces in a terminal-independent manner (GNU Project, (2023)) – was attempted. It provided smoother graphics and better control over GUI element positions. However, ncurses wasn’t included in the default Windows C library installation as it was written for Linux in 1993, and installation and compilation proved to be challenging. Though it was successfully tested, deploying it required dynamic linking via .dll files for systems without the library, which was beyond the scope of this project due to time constraints.

Monster pathfinding with breadth-first search was another steep learning curve as it came with implementation of queue data structures and learning how the algorithm worked. It worked as intended. Current implementation of monster AI lacks intelligence as it can track through walls and only strictly follows the player without other functionality. Future versions can include separate behaviour defined for different type of monsters, such as bats having echolocation and being able to detect through walls and running away when they have low health.

Combat works as intended. However additional features such as ranged attacks, spells or additional combat mechanics can improve combat gameplay and strategy.

The base game engine is pretty much complete and works well to create games, however due to time constrains, proper levels haven’t been played tested and designed, with unbalanced items and a limited variety of monsters and consumables. Though items have been implemented in code, there is no way to pick up items, weapon and armour in game after killing monsters or finding treasure, etc, and no way to spend gold has been implemented.

# Conclusion

DUNGEON.txt a modular ASCII terminal-based dungeon crawler was implemented and provided experience in using the core principles of C programming, namely data structures, dynamic memory allocation, text file parsing, header files and handling player input and output.

This project has huge room for improvement and will have future versions fixing current issues such as crashes from file formatting errors and implementing smarter AI and a more cohesive combat system. Levels, items and proper enemies should be designed and play tested for a more challenging and immersive experience.

# References

- Ford, A. (2018). *Shadowcasting in roguelike games*. [online] Available at: <https://www.albertford.com/shadowcasting/> [Accessed 13 Apr. 2025].

- GNU Project (2023). *ncurses - GNU Project - Free Software Foundation*. [online] Available at: <https://www.gnu.org/software/ncurses/> [Accessed 13 Apr. 2025].

- Kernighan, B.W. and Ritchie, D.M. (1988). *The C Programming Language*. 2nd ed. Englewood Cliffs, N.J.: Prentice Hall.

- Kitware (2025). *CMake*. [online] Available at: <https://cmake.org/> [Accessed 13 Apr. 2025].

- Kundu, S. (2020). *Breadth First Search (BFS) Algorithm*. GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> [Accessed 13 Apr. 2025].

- Sebesta, R.W. (2016). *Concepts of Programming Languages*. 11th ed. Boston: Pearson.

- W3Schools (n.d.). *Data Structures and Algorithms - Linked List*. [online] Available at: <https://www.w3schools.com/dsa/dsa_theory_linkedlists.php> [Accessed 13 Apr. 2025].

- Wizards of the Coast (2014). *Player’s Handbook*. 5th ed. Renton, WA: Wizards of the Coast.

# Appendices

## Appendix A: Room text file example

maze\_room\_2.txt from assets/rooms

ROOM NAME: Maze

ID: 6

DIMENSIONS: 12,12

############

#..........#

####.#####.#

#..........#

#.##########

#..........#

####.###.###

#......#.#.#

#......#.#.#

#......#.#.#

#..........#

############

ENTITY COUNT: 3

MONSTER FILE: bat.txt ; GRID POSITION: 10, 7 ; CURRENT HP: 2

MONSTER FILE: bat.txt ; GRID POSITION: 5, 8 ; CURRENT HP: 2

MONSTER FILE: bat.txt ; GRID POSITION: 9, 1; CURRENT HP: 2

START DIALOGUE: maze\_room/start.txt

CLEAR DIALOGUE: maze\_room/clear.txt

ENTER DIALOGUE: x

EXIT DIALOGUE: x

DOOR COUNT: 1

DOOR: GRID POSITION: 1, 1 ; TARGET ROOM ID: 1 ; TARGET ROOM POS: 5, 5

## Appendix B: Entity text file example

NAME: goblin

TYPE: 1

SYMBOL: G

MAX HEALTH: 8

ARMOUR: 12

SPEED: 15

INITIATIVE MOD: -3

ATTACK: 2

DAMAGE: 5

GOLD REWARD: 15

## Appendix C: Item text file example

NAME: Rusty sword

ATTACK: 2

DAMAGE: 4

## Appendix D: Dialogue text file example

[Press any key to continue dialogue]

\*THUD.\*

…What was that?

{wait}

Wh—Where am I?

{username}…

{wait}

That name… Is that mine?

{wait}

\* {username} slowly blinks against the darkness. They lie half-buried in a cold pile of rubble, the air thick with dust and silence. Only thin shafts of light pierce through cracks in the stone above. \*

{wait}

\* As {username} pushes the debris away and rises, an icy voice whispers through the gloom, threading chills down their spine. \*

{wait}

You have been summoned…

…to face the Trials of the Blade.

{wait}

\* A pale, spectral figure fades into view, suspended in the air before you. Its eyes — hollow yet piercing — lock onto yours. \*

{wait}

Steel yourself, lost one. The path ahead will test your spirit.

Take this.

{wait}

!!! {username} receives a \*\*Rusty Sword\*\* !!!

\*The weight of destiny presses into your palm.\*

{wait}