C++ project documentation -Dungeon Crawler group 3

Overview:

The project is a semi-procedurally generated roguelike turn-based dungeon crawler with heavy emphasis on tactical approach and thinking every move carefully. In a sense, every room of the dungeon is a semi-procedurally generated puzzle.

Features:

*Six semi-procedurally generated levels. Each level, except the first one, consists of a 3x3 grid of nine rooms, which are generated from randomly picked room files. Level shapes are fully randomized by a recursive ”walker” algorithm. Rooms are generated from hand-made text files, but they are randomly mirrored vertically and/or horizontally to create more variety, and trap states are also randomized. Enemies that spawn in rooms are randomly picked from a selection of hand-made enemy vectors and spawned on tiles defined in the room text files. The amount of loot and the quantity of different loot types is always the same, but loot items inside categories like melee weapons may vary – a mace in one run can be a sword or an axe in another. Loot is randomly distributed on the levels in randomized rooms.

*Permadeath and increasing difficulty as the player clears levels. Difficulty is created by spawning in harder enemies depending on how deep the player has descended into the dungeon.

*Win condition – reach and clear the sixth level, and defeat the mini boss at the end.

*Losing condition – lose all health points.

*Turn-based gameplay where the player moves first, followed by trap tiles and then hostile entities.

*Tile-based movement of the player and enemies from a top-down perspective.

*The player character is a cube, and the different sides of the cube function as inventory slots. Movement from a tile to another rolls the cube from one side to another, and the item that ends on the top of the cube is used automatically if it can be used, e.g. an enemy is in range of a ranged weapon and the weapon isn’t on cooldown.

*The player can freely assign owned items on the sides of the cube they want and drop unwanted ones. Swapping items between slots is also possible. Inventory (the cube surface layout of items) can’t be accessed during combat.

*Items can be picked up by moving over them, and they will be placed in the first free inventory slot available and will be on a short cooldown before they can be used after having been picked. Dropped items will have a short cooldown before becoming pickable again. If the player has full inventory, items on the floor are ignored. Several items can’t exist in the same tile, and dropping an item on top of another one is not allowed. Thrown weapons will bounce onto the nearest available tile if they are thrown on enemies that are standing on a tile that already has an item.

*Enemies that look like cubes but will follow a simple pathfinding depending on their type – either aggressively towards the player or staying at a safe distance to use ranged attacks. Enemies will also proceed through an item usage pattern unique to their type instead of rolling from one side to

another like the player, and their index in this vector of actions is randomized when enemies are generated.
*Trap tiles that have three states that they cycle through every turn – dormant, emerging and spikes. Standing on a trap that pushes its spikes up will damage the character standing on it.

*The game is played with WASD / Arrow keys and mouse controls. Full mouse gameplay is also supported.

*No line of sight. Players can see everything in their current room. Enemies will be able to see the player after they step into the room until defeated.

*If a room has an enemy vector that has enemies in it, the doors of the room will be locked when the player enters the room, and will only open after all of the enemies have been defeated. If the room has a loot item assigned to it, the item will be spawned when the doors open or in case there were no enemies – immediately when the player steps into the room.

*Automated detection for softlocks e.g. the player can't move due to being completely surrounded by tiles that can't be moved to due to enemies or obstacles occupying them. This will simply skip the player's turn.

*A map button that opens the map of the current level.
*An inspect button that opens an information window of the selected item or enemy.
*A button for activating inventory interactions such as moving items.
*A button for returning to main menu.
*Buttons for moving with mouse.

*Main menu that has three buttons – One for quitting the game, one for opening a text tutorial and credits, and one for starting a new run.

*Indicators for health and dungeon level, as well as constantly shown information about item states such as weapon damage, current cooldown and the like.

*Self-made pixel art graphics.

Software structure:

main.cpp:
Main – Opens the main menu window for the game.
LevelLoop - and when a game run is started it holds a loop that proceeds the level counter and initializes new levels and goes into the actual game loop of playing a single level every cycle. tracks keypresses from the player and keeps the game going until the player character's death or if the current level has been completed.
Level – Holds an instance of a single level. Asks for player input and cycles through character and trap turns, and calls graphics updating functions appropriately.

Storage files:
RoomStorage folder – Holds the different room shapes for dungeon generation. Rooms are stored as ASCII art of 12 x 12 grid with different characters representing enemy spawn locations, trap locations, walls and the like, which are translated to level features when the file is read when a

DungeonRoom is instantiated. Rooms will also be able to be rotated into any orientation and mirrored by the level generation algorithm to create more variety of a single pre-set

room shape.

Graphics – Holds the pixel art sprites for the visual representation of the game.

Main classes:
DungeonTile – A class that holds information about the contents and the state of a tile on the game board and its type, such as empty floor tiles, doorways, walls, traps and pits.

DungeonRoom – A class that represents a room consisting of 144 DungeonTiles. Rooms randomly select their layout from the aforementioned RoomStorage file, and are connected to other rooms by Door type DungeonTiles. Also holds information like a loot item and has methods for additional actions such as spawning enemies, closing doors and opening them.

DungeonLevel – A class that holds and connects nine DungeonRooms together on a grid to create levels. Only one level exists at a time, and when a level is completed it is deleted and a new one is created. Level shapes and player starting location in a level are randomized.

Item – A virtual class that all item classes inherit. Holds information about the item state such as cooldown, durability, name and description.

> *Subclasses: various abstract classes for different item types such as melee & ranged weapons, shields, spells and potions, which will then be inherited by item classes inside those types. For example, a sword inherits the melee weapon class, which inherits the item class.

InventorySlot – Holds an instance of the Item class in it, empty by default. The item in an InventorySlot is used if it isn't on cooldown, the InventorySlot is on the top position of the player cube character and the conditions for using that particular item are met, for example an enemy is in range of a weapon or an enemy is attacking the player while a shield is on the top of the cube. Functions as the interface for using items and accessing information about items in the player's inventory.

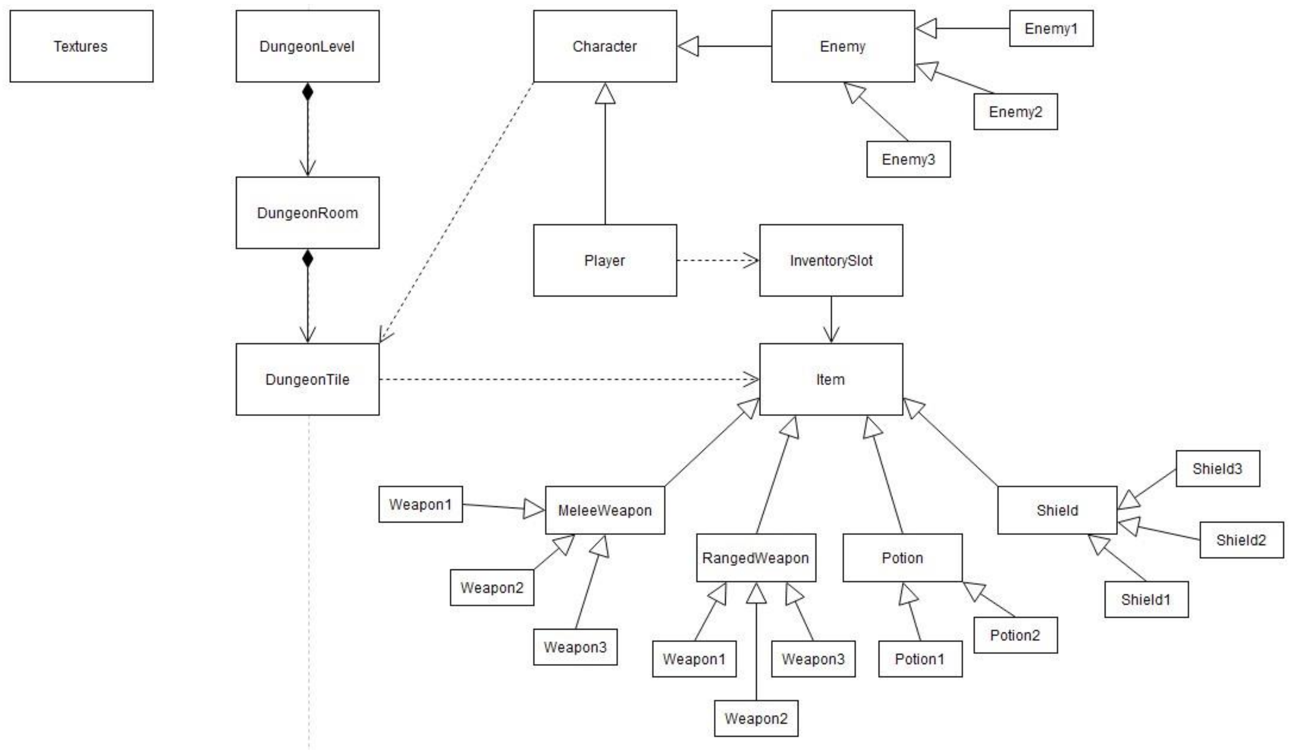
Character – A virtual class that Player and Enemy classes inherit. Holds information about the state of the character such as health points, action vector, character type and name. *Subclasses: Player, Enemy

Player – Holds information about the status of the player such as inventory. The orientation of the player cube character and the inventory are stored in a vector of six InventorySlot objects, which are moved inside the vector to change the orientation of the cube depending on player movement.

InventorySlot – Holds an instance of the Item class in it, empty by default. The item in an InventorySlot is used if it isn't on cooldown, the InventorySlot is on the top position of the player cube character and the conditions for using that particular item are met, for example an enemy is in range of a weapon or an enemy is attacking the player while a shield is on the top of the cube.

Enemy – A virtual class that all enemy classes inherit. Holds special methods such as pathfinding towards the given character and proceeding through its action vector. *Subclasses: All different enemy types.

Textures – A class that is instantiated only once when the game is run, and it stores all sprites needed in the game to avoid loading the sprites from disk every time they are needed. This optimizes the game performance significantly, and passing a pointer to the once instantiated Textures object takes this optimization even a step further.



Instructions for building and using the software:

Project folder 'out' includes a ready Linux build of the program which can be run straight out of the box. The folder includes an executable 'DungeonCrawler' for Linux. SFML library is included statically. The program can be run on Linux systems as follows:

1. Go to folder `out` using terminal
2. Run the executable by typing `./DungeonCrawler`

Because of static SFML inclusion, the program doesn't have to be compiled locally to run it. If you want to recompile the program anyway, you can do it as follows:

1. Empty the contents of folder `out`
2. Go to folder `out` using terminal
3. Type command `cmake -S .. -B .`
4. Type command `make`
5. Once the program has compiled, run the executable typing `./DungeonCrawler`

Testing:

Dungeon generation related classes were tested with test scripts that generated one instance of each class being tested, printed information about it to standard output, changed some values and printed information again in order to see if the classes functioned properly. First class to be tested was DungeonTile, as it was needed to work flawlessly for the DungeonRoom class, which was tested after DungeonTile by generating a DungeonRoom and then printing all its DungeonTiles as characters to the standard output. Character classes (Player and Enemy) were tested using a single instance of the DungeonRoom class by printing different DungeonTiles and Characters with various letters and characters to see their state and movement. After the Character objects managed to move in a single room, the DungeonLevel class was tested. A single instance of the class was created, and all of the DungeonRooms inside it were then printed as text characters to the standard output, allowing viewing how rooms connected to each other with their doors.

Item classes and the player character movement were tested after the main game loop was in a state where it generated one level and drew the graphical presentation of it using proper pixel art sprites for easier viewing and testing. The few bugs there were easy to track down due to very specific conditions where they appeared and the ease of reproducing the issues. Segmentation faults were tracked with the debugger or test printing during the execution of the code, and in all cases were caused by trying to access the methods of a class, whose instance being handled was a null pointer.

After the main game loop was ready for the player to traverse from a level to another, thorough playtesting was done to fix the last small bugs in enemy pathfinding, typos in the hand-made room text files and to tune the difficulty to be challencing enough for it to be difficult for the developers of the game to beat it.

Work log:

Summary of responsibilities:

Elias Viro - Game and UI design, designing the main game mechanics, enemies, items, rooms and project structure, balancing the game, creating pixel art sprites and guiding the team.

Atte Aarnio – Implementation of the graphics, as well as interfaces with the graphics libraries and configuring the project and tools.

Selin Taskin – Programming the enemy movement algorithm and creating the basis of item classes.

Binh Nguyen – Creating the level generation algorithm.

Working schedule. Project meetings are not accounted in the amount of hours spent on the project.

Week 43 (week 1 of the project):
Elias Viro: Finished the project plan and the game engine related diagrams, concept art and class structure. Time usage estimate: 5 hours.
Atte Aarnio: Studied SFML integration into the project and created a diagram of class relationships on the graphics side. Time usage estimate: 5 hours.

Selin Taskin: Read through the project plan. Time usage estimate: 1 hour.
Binh Nguyen: Read through the project plan. Time usage estimate: 1 hour.

Week 44 (week 2 of the project):
Elias Viro: Implemented the file structure, created some base classes and created instructional comments for some classes. Configured SFML. Time usage estimate: 12 hours.
Atte Aarnio: Got SFML working and advised other project members how to configure it. Time usage estimate: 15 hours.
Selin Taskin: Started the groundwork for some item classes and studied the instructional comments and structure of the project. Time usage estimate: 3 hours.
Binh Nguyen: Worked on other courses. Time usage estimate: 0 hours.

Week 45 (week 3 of the project):
Elias Viro: Implemented some item classes, finished Character and Player classes, fixed tons of errors, started creating sprites for the game. Time usage estimate: 16 hours.
Atte Aarnio: Worked on SFML and other graphics engine related things, managed to get the warnings to show for the project. Time usage estimate: 16 hours.
Selin Taskin: Continued working on some item classes. Time usage estimate: 3 hours. **Binh Nguyen:** Started the implementation of the DungeonLevel class and the level generation algorithm. Time usage estimate: 6 hours.

Week 46 (week 4 of the project):
Elias Viro: Created more pixel art sprites for the game, added a bunch of enemy classes, debugged lots of errors and implemented loot and enemy vector randomization functions for the main game loop. Time usage estimate: 15 hours.
Atte Aarnio: Configured CMake and the debugger, created a functioning main menu screen and laid the groundwork for main game loop. Time usage estimate: 15 hours.
Selin Taskin: Started the work on enemy pathfinding algorithms. Time usage estimate: 8 hours.
Binh Nguyen: Fixed compilation errors related to the DungeonLevel class and the level randomization, added a few rooms in RoomStorage for testing purposes. Time usage estimate: 5 hours.
Week 47 (week 5 of the project):
Elias Viro: Created more pixel art sprites for the game, debugged and fixed a bunch of errors and segmentation faults, implemented the core structure of the main game loop, playtested the game, helped with the enemy pathfinding. Time usage estimate: 27 hours.
Atte Aarnio: Implemented game graphics printing functions and interactive buttons and other menus. Time usage estimate: 30 hours.
Selin Taskin: Implemented the enemy pathfinding algorithm. Time usage estimate: 10 hours.
Binh Nguyen: Fixed remaining bugs in the DungeonLevel class. Time usage estimate: 2 hours.

Week 48 (week 6 of the project):
Elias Viro: Playtested the game, fixed remaining bugs in the project, helped with button interactions, created last missing sprites. Time usage estimate: 22 hours.
Atte Aarnio: Implemented menus and button reaction actions for all of the buttons. Time usage estimate: 22 hours.
Selin Taskin: Worked on other courses. Time usage estimate: 0 hours.
Binh Nguyen: Worked on other courses. Time usage estimate: 0 hours.