Project documentation
Dungeon Crawler Game 2

1. Overview

The game is a roguelike dungeon crawler inspired by games such as Binding of Isaac and traditional pen and paper roleplaying games. The game features real-time combat, randomly generated rooms and collectibles and a progression system in which the player's score increases as they delve deeper into the dungeon facing more and more difficult monsters and rooms to fight through.

If we take a look at our initial plan for features, most of them are in place in the final game. Character progression isn't as vast as we planned and shops were phased out of development. But the end product still has all the core features in place.

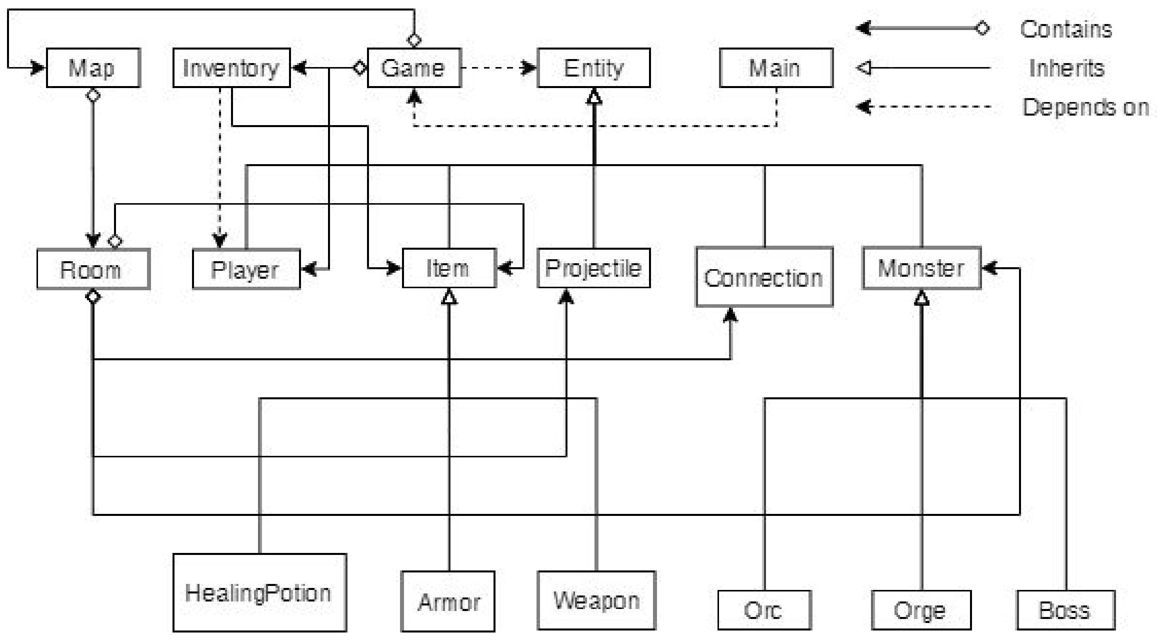
Basic features

- Simple 2D graphics: Done
- Moving through doors: Done
- Combat: Done
- Collectible power-ups: Done
- Scoring system: Done

Additional features

- Random generation of maps, monsters, and collectibles: Done
- GUI: Done
- Character progression: Not quite, but there's a difficulty system. The game keeps getting progressively more difficult each floor until the player loses
- NPCs (Shopkeepers, etc.): No
- Sound effects, music: Partially

2. Software structure



UML diagram of the program.

The main function is only dependent on a Game object. Each time a game is launched, a new Game object is created. The Game class models the concept of a dungeon game session. Inventory represents the player’s inventory, managing different items that the player can utilize.

Game is a class representing the singular game session and contains all other objects appearing in the game. When called, the Game constructor loads a fresh instance of the game and displays it in the SFML renderwindow provided to it as a pointer. Game generates a Map using the generator method of that class. Most Entity creation is handled in the Map module, Game just stores the outcome. A new Player object is also created, representing the player character. An Inventory object is created to handle Items held by the Player. Once the initialization is done, the main function calls the different parts of the game loop until the game is closed, determined by a boolean in the game object. Game directly takes care of handling user input by creating the projectiles the player attacks with and adjusting player acceleration, as well as indirectly handling updating and rendering all objects in the game by calling their individual methods. When the Player dies, the Game stops updating, and draws an additional game over text.

Entity is an abstract class representing the concept of an entity that can exist in a Room, has a coordinate within a Room, and can have a velocity. Entity has basic setter and getter methods. No moving Entity except for the player can escape from the room.

Player inherits from Entity and has hit points (HP) and a score. A Player can accelerate and decelerate and move around by pressing WASD and can attack monsters. The Player gains scores by killing monsters. After being attacked by a Monster, the Player is immune to damage for a few frames. A Player stores a Room pointer which points to the player’s current Room. It also stores a pointer to the Inventory to facilitate interaction between the player and the inventory.

Monster inherits from Entity and has hit points. Each Monster has a sprite and is responsible for managing its sprite. A Monster can be either active or inactive. If it is killed by the player, it will become inactive and will not be rendered. On spawn, a Monster automatically adjusts its spawn such that its coordinates will not be outside the boundaries of the Room. A Monster also stores a pointer to the Player in order to determine its behavior. A monster also accesses the player's current room via the Player pointer. Different Monsters have different characteristics. Orc is the most basic kind of monster, having the lowest hit points, dealing the least damage, and having very simple behavior. It moves from (left, top) to (bottom, right) at a constant speed. Orge is a more advanced monster. It moves towards the player at increasing speeds. Boss represents the boss of each level, having the highest HP and most advanced AI. It has two stages depending on its remaining HP. When a Boss has lots of HP, it uniformly runs in circles and regularly shoots out volleys of hard-hitting projectiles toward the player. When its HP gets low enough, it enters an "enraged" mode, in which it charges towards the player, temporarily stops when it reaches the player, and charges again. In this mode, it does not shoot projectiles but deals very high damage upon contact with the player. Monster damage and hit points are based on difficulty. The higher the difficulty, the higher the damage the monsters deal and the more HP they will have.

Items inherit from the entity class. They are objects in the dungeon the player character can find in item rooms and pick up by walking over them. Items are stored in an inventory class to where the player has access. The inventory can hold (own) a single item of every type (armor, weapon consumable) at a given time. The player drops behind the held item if the picked up item is the same type. Weapons simply affect the player's damage output; armor reduces the incoming damage; and consumables are one time use items. In particular, the healing potion will heal the player up to full HP when the E key is pressed.

Room represents a single room in the dungeon. Almost every class interacts with Room for their activities. Almost all activities happen within the player's current room. A room owns projectiles, monsters, and items. Each time a projectile is fired either by the player or a Boss, it is added into the internal collection member of the object. This is also the case with monsters and items. Room has basic getter and setter methods for other classes to interact with the elements inside Room. Rooms are connected to each other by having pointers to their cardinal neighbors. The number of rooms within the map is based on the game's difficulty. The higher the difficulty, the more rooms there are. Most rooms contain a handful of randomly placed monsters but there are also special rooms. Special rooms include the boss room, which as the name suggests houses the floors boss and the item room which randomly spawns equipment and consumables to increase the players strength. Traversing from one room to another is handled with Connection objects that can be considered as doors. A Room owns several Connections. Doors unlock once the room that they're in has been cleared of monsters. As the player interacts with the door, depending on its facing, the player is sent to the corresponding neighboring room.

A map object represents a floor of the dungeon. Each map owns rooms. Upon creating a map rooms are created and assigned neighbors to create a random network of rooms. The map object initializes its rooms and puts in each room a number of entities based on the difficulty. The higher the difficulty, the more monsters and items will spawn in each room.

Memory management has been implemented with unique pointers. In particular, in this program, an object “owning” another means that the owning object has as its member a unique pointer pointing to the owned object. Room owns Projectiles, Monsters, and Connections. Map owns Rooms. Game owns Map, Player, and Inventory. Inventory and Rooms share Items between each other. When an Item is picked up by the Player, ownership is transferred from Room to Inventory. When the Player drops an Item, ownership is transferred from Inventory to Room. In this way, only 1 object can own an Item at any time. Unique pointers are used because it is easy to keep track which objects own which and thus responsible for memory release. However, there are still memory leaks which are caused by the underlying SFML library, not the code of the program.

3. Instructions for building and using the software

Installation

1. Download the repository
2. Run ***sudo apt-get install libsfml-dev*** on WSL to get the SFML-library required. If not on a Linux-based on system, look up the relevant tutorial on SFML website to install.
3. Run **cmake .** in the project directory
4. Run ***make*** in the project directory to compile the project
5. To display the graphics install Xming or equivalent (**Only do steps 5-7 if working remotely or on WSL**) (https://sourceforge.net/projects/xming/)
6. Run Xming when a new terminal is launched.
7. Run ***export DISPLAY=:0***
8. Run ***./dungeons*** to launch the game
9. Note: if music does not play upon launch and there are run-time errors (because of the local system), comment out the following code in main.cpp:

```
sf::Music music;
if (!music.openFromFile("src/sprites/main.ogg"))
{
    std::cout << "music error" << std::endl;
}
music.play();
music.setLoop(true);
```

Playing the game

The player character is controlled with WASD-keys, press Mouse1 shoots a projectile and E to use consumables.

The objective of the game is to clear rooms of enemies to progress towards the boss encounter and to gain as many scores as possible by killing monsters. The player gets progressively stronger as they pick up items in item rooms. After defeating the boss the player can walk into a portal in the boss room to enter a new floor with more monsters and better loot. The game goes on until the player dies; and as difficulty ramps up with each new floor generated, the number of monsters and rooms generated increases.

4. Testing

Testing of Game and Player control

Testing of the main game loop was possible in a very early stage, since few modules were needed, meaning also very short compile times. No advanced testing tools were required, whether the game posted correctly was more than enough for our purposes. Player inputs were also easy to test, since there were similar requirements for the depth of testing. More testing was needed for determining the correct functionality of player controls. Getting the player acceleration to add up correctly took some fiddling to get right, since initially the velocity calculation formulas were a little off. Once again, simple print-tests and playtesting were enough for finding out and fixing major issues with player behavior. Automated testing would have probably saved a lot of work at times, since some bigger problems required commenting out entire modules.

Testing the map and its modules

Since the map generation was in place before room connectors testing out whether rooms were generated randomly and their neighbors were set properly was mostly done by printing debugging information to the console. Most problems arose from assigning the neighbors and checking the right cells in the helper layout. But assigning numbers to each compact direction and making sure the numbers matched on each part of the generator made tracking down the issues easier.
Once simple graphics were in place connections between doors could be tested. Again printing into the console when something worked or didn't was the main way to track down issues. On testing the connections it was found out that there was an issue in the map generation that the neighbors weren't assigned properly. After more printing the issue was tracked down and resolved after which maps, connections and rooms all worked.

Testing of items and inventory

Items were tested thoroughly once the whole item structure was done around the second and third week once the player movement was done. Item dropping was the main focus of the testing because there was the biggest chance that something could go wrong. Items were tested again once all of the branches were merged to master. The merge introduced more variables and more problems. Item collision, item creation and consumable got finalized during the final week of the project so everything worked accordingly.

Testing of monsters and their AI

Most if not all problems came from the AI of each kind of monster. After the behavior of a monster was defined, it was tested manually by playing the game and seeing if the behavior is as expected. Printing on screen was the primary method to inspect the state of the program, although sometimes GDB-debugger was used to track down segmentation faults and other unexpected behavior. Most problems came from wrong calculations. Corresponding adjustments to calculations were applied and gradually bringing the AI to the expected behavior.

Testing the damage system

Because this is mainly related to the values of the variables within the program, printing on screen was the main method used for testing. All values related to damage, such as player HP and monster HP were printed out. Most problems came from the fact that monsters and the player were not updated correctly. Once, it was also found that the items had no effect on the player or the damage dealt to monsters. This was easily fixed by fixing a few parameters and variables in the update methods of the player class and monster classes. Collision did not work correctly before because it was manually checked by user code. It was discovered that the SFML sprites support border intersect checking so that method was used to check for collision between different entities. Automated testing would have helped here, but it was not an issue since the damage system only consist of a few lines of codes, thus taking little effort to fix.

5. Work log

The division of work laid out by our project plan was as follows: Son handles the entities and the player, Rashid map generation, Antti items, Leo the game loop and event handling, and everyone would work on the graphics of their own modules.

Week 1 (19.7 - 25.7)

Antti: Reading on sfml and setting it up. Testing different things on sfml. Creating a draft of the item system. (3h)
Leo: Trying to get SFML to work. Planning the software structure (5h)
Rashid: Getting familiar with using github. Trying and failing to get SMFL to work. (3h)
Son: Getting familiar with git, trying to get SFML to work. Researching from various sources on game design and deciding how best to implement entities. (4h)

Week 2 (26.7 - 1.8)

Antti: Creating the first version of the item system including the inventory, different types of items and the base item entity. (8h)
Leo: Main loop, game class, and more planning. Test version complete early next week (6h)
Rashid: Very basic headers for map and room. Researching different ways to randomly generate maps (4h)
Son: Very basic header and implementation for Entity. Reviewing C++ and testing various things with SFML. (5h)

Week 3 (2.8 - 8.8)

Antti: Created a second version of the item classes, but it didn't pan out nicely so I scrapped it. Fixing and testing the first version for the rest of the week. (8h)
Leo: Debugging and fixing coding errors in the test version of the game. Writing player controls and projectile handling. (5h)

Rashid: Adding basic interfaces such as getters and setters to rooms. Considering different ways to connect rooms. (3h)
Son: Implementing Player with Leo. Trying to get Player to move properly. Deciding on what kinds of monsters and their AI to implement. (5)

Week 4 (9.8. - 15.8)

Antti: Didn't get much done other than some fixes. Planned how the player drops an item and how the player can't drop the item out of bounds. (3h)
Leo: Slow week, minor improvements to existing code while waiting for other modules. (2h)
Rashid: Settled on a way to handle the map generation. Started writing down logic for the generating. Considered ways to read room layouts from a text file. Scrapped the idea since it was too time consuming. (6h)
Son: Started implementing various kinds of monsters. Fixing errors from failed merges and helping others with their work. (4)

Week 5 (16.8 - 22.8)

Antti : Added item drop function, healing potions and some minor features. Tested the item collision method thoroughly with sprites. (10h)
Leo: Fixing coding mistakes, troubleshooting and solving compatibility issues between modules. (4h)
Rashid: Finished logic for map generation, did a bunch of debugging until everything compiled and worked as it should. Changed the way connections work to avoid circular dependencies. (6h)
Son: Started working on memory management. Finishing up monsters. Added interaction between monsters and the player. (6h)

Week 6 (23.8. 28.8)

Antti: Combining items with the master branch and fixing all of the errors and bugs brought up with the merging. Refining the item system and adding some smaller features and fixes such as onscreen armor and damage values. (15h)
Leo: Fixing coding mistakes. Implementing graphics, fine-tuning multiple gameplay features. Converted the entire game to use smart pointers and cleaned up code. Considered adding sound effects to the game, but couldn't get audio to work. (20h)
Rashid: Added special rooms to map generation. Finished random placement of monsters in rooms. Properly implemented connections, so the player could traverse from one room to another. (15h)
Son: Finishing up entity classes. Finalizing interaction between the Player, Monsters, and Items. Made code look prettier. Trying to make the graphics more beautiful but failed. Adding comments. Fixing obscure bugs. (14h)