Project Specification - Roguelike RPG

Ying Stokes

October 2020

1 Introduction

This project is a Rogue-like Role Playing Game implemented using the ncurses library for graphical output. It allows the user to control an adventurer to explore a dungeon. The dungeon should contain procedurally generated rooms which contain similarly procedurally generated entities of interest, such as monsters and items.

2 Design Description

2.1 Assessment Concepts

2.1.1 Memory Allocation from the Stack and the Heap

- **Arrays:** The game map will be implemented using a two-dimensional array.
- **Strings:** Most objects such as monsters, items, and parts of the environment will have textual information associated with them, such as descriptions of enemies.
- **Objects:** The project will rely heavily on object-oriented design, so most functionality of the game will use objects, such as entities, items, levels, the renderer, etc.

2.1.2 User Input and Output

- Input/output will be implemented in a blocking fashion (as the game is turn-based) to retreive user input commands in the form of particular keypresses (or combinations thereof).
- This will be achieved using ncurses, which will parse user input per-character, and alter the game state based on a number of factors (such as menu contexts etc.).

2.1.3 Object-oriented programming and design

- Objects will be used to model all aspects of the game environment, such as objects representing terrain tiles, and game entities.
- **Inheritance:** In order to implement various types of animated entities in the game, such as the player, as well as monsters that might oppose the player, a parent class to model a living creature will be used.
- **Inheritance:** Items in the game must all be able to do certain things, such as getting picked up by the player, or being dropped, or used. Thus, all items must inherit a base "item" class, while each individual item type may incorporate unique behaviours.
- **Polymorphism:** Items in the game will exist as one of multiple subtypes, such as equippable items, generic items, and comestible items, which will be implemented using another **abstract** class which will include a virtual function for the consumption of that item.

2.2 Testing

2.2.1 Unit Testing

Unit testing will be used to verify that individual class methods are functioning corrently through the calling of getters and setters in a separate c++ program. For example, the following code will test an entity's movement.

```
Entity entity(
    "Test Entity",
    "This is an entity instantiated for the purpose of testing the Move(int y, int x) function.",
    0,
    0,
    'T');
entity.move(0,1);
if (entity.GetX() == 1) {
    std::cout << "Entity::move(int y, int x)...... OK" << std::endl;
} else {
    std::cout << "Entity::move(int y, int x)...... FAILED! Entity::GetX() returned " <<
        entity.GetX() << std::endl;
}
```

2.2.2 Integration Testing

Some of the game's processes will not be easily testable using unit testing, so integration testing will be used to verify that elements of the user experience are funtioning correctly.

For example, the following integration test will be used to exhaustively verify that orthogonal movement will be perceived to function correctly:

```
[Game Start]
[Player presses up arrow]:
    Player Entity's '@' character moves up on the screen by one character.
    A debug message appears in the HUD indicating that the player moved up.
[Player presses down arrow]:
    Player Entity's '@' character moves down on the screen by one character.
    The previous debug message moves up to make space for the next.
    A debug message appears at the bottom of the HUD indicating that the player moved down.
[Player presses left arrow]:
    Player Entity's '@' character moves left on the screen by one character.
    All previous debug messages move up to make space for the next.
    A debug message appears at the bottom of the HUD indicating that the player moved down.
[Player presses right arrow]:
    Player Entity's '@' character moves right on the screen by one character.
    All previous debug messages move up to make space for the next.
    A debug message appears at the bottom of the HUD indicating that the player moved right.
```

2.3 Class Descriptions

2.3.1 Entity

This will be a class describing a physical object within the game, excepting terrain. This object will be used to represent both animated entities, such as monsters and players, as well as inanimate objects, such as items. This object stores the health of the entity, as well as various properties including name and position.

Base Class: The base class will describe a generic entity, which will be used to create inanimate objects in the world such as boulders.

Player: The player is one of two types of animate entities in the game, of which this one will be controlled by the player. This class differs from the base entity in that it keeps track of inventory including weight.

Enemy: Similar to the player, this represents the other type of animate entity. This class differs from the base entity in that it keeps track of an inventory (loot), and contains methods to drop the loot and attack the player.

ItemEntity: A unique type of entity, this represents a physical item in space, able to be picked up by an animate being in the world. Thus, this object tracks the corresponding *Item* object, and contains methods to be picked up.

2.3.2 Item

This class represents an item as it exists within the storage some other entity, and thus includes information about the item's properties such as value and weight.

Base Class: The base class will simply contain the item's weight and value information, thus representing the most basic of items, such as money or rocks.

EquippableItem: The *EquippableItem* subclass represents an item that can be equipped by another entity, and thus includes variables that track which item slot it is relevant to, and the buffs provided by the item.

ComestibleItem: This subclass is an abstract class representing an item that can be consumed by an entity, and thus includes a virtual function for the effects upon consumption.

HealingItem : ComestibleItem: The *HealingItem* is an example of a *ComestibleItem* which would implement a Consume method that would heal the user.

2.3.3 Level

The *Level* object will represent a level of the game world. Thus, the class includes member variables for storing entities and environment information relevant to a particular level.

2.3.4 Room

The *Room* object contains the information relevant to a single room within the world, and includes methods for calculating useful information about the room, such as its area.

2.3.5 Tile

Tile is a struct that contains the information relevant to a single type of tile within the world, such as a floor or a wall.

2.4 Class Diagrams

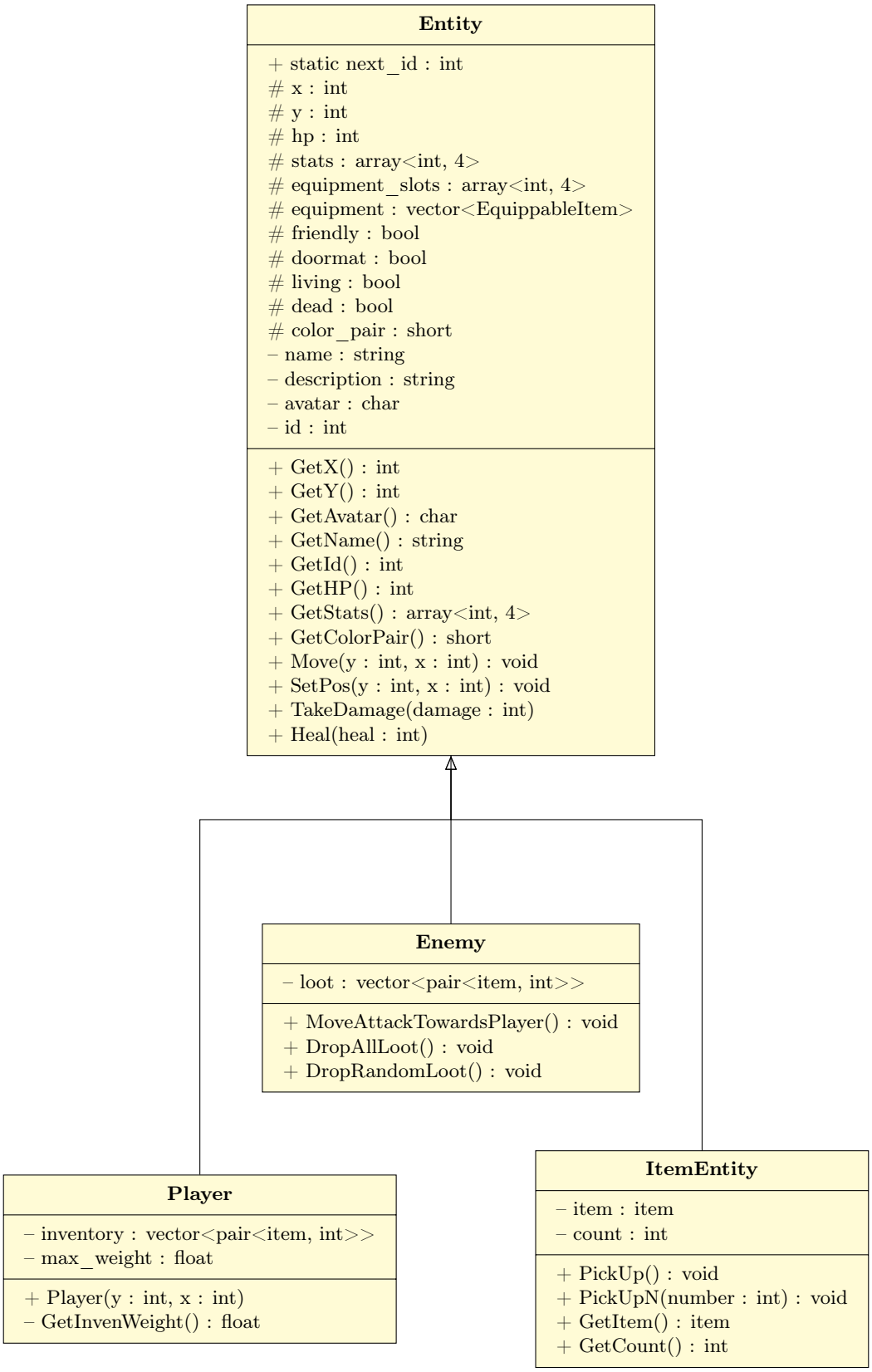

Figure 1: Entity Class Diagram

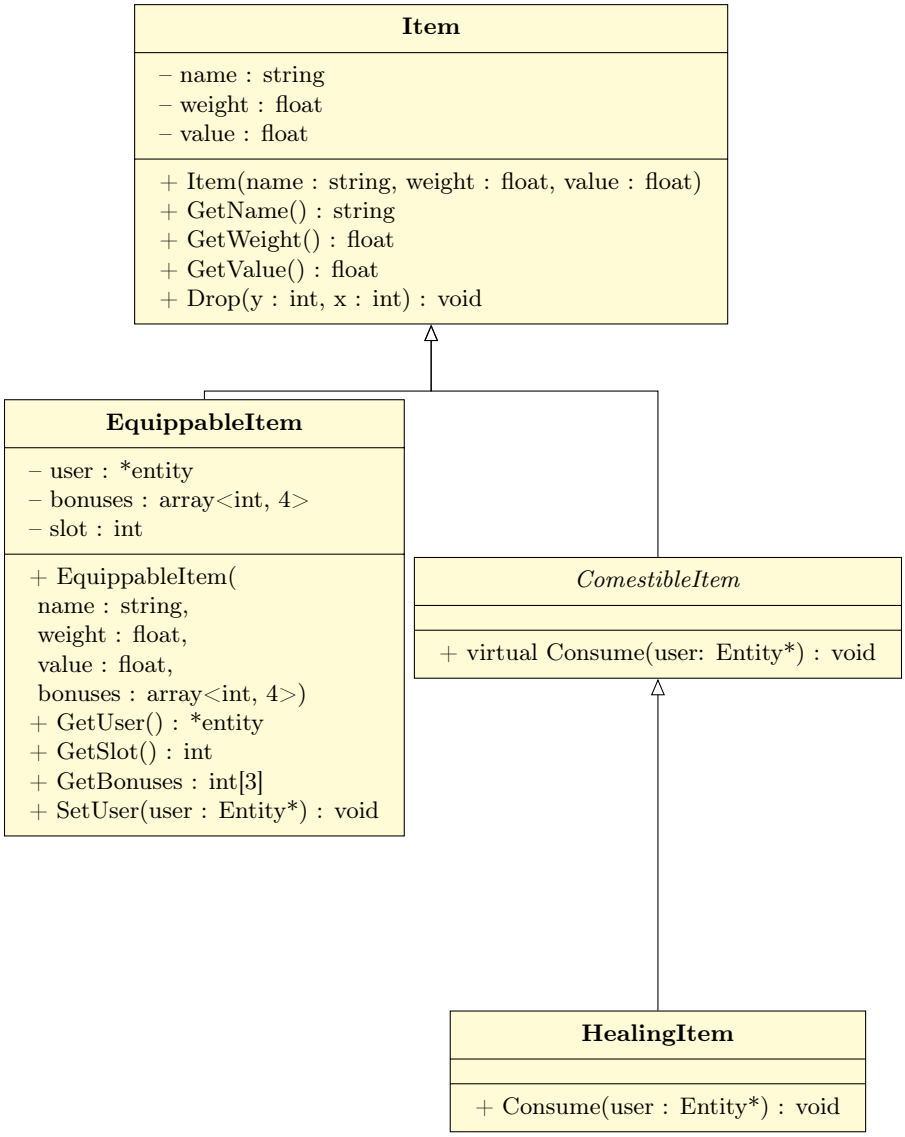

Figure 2: Item Class Diagram

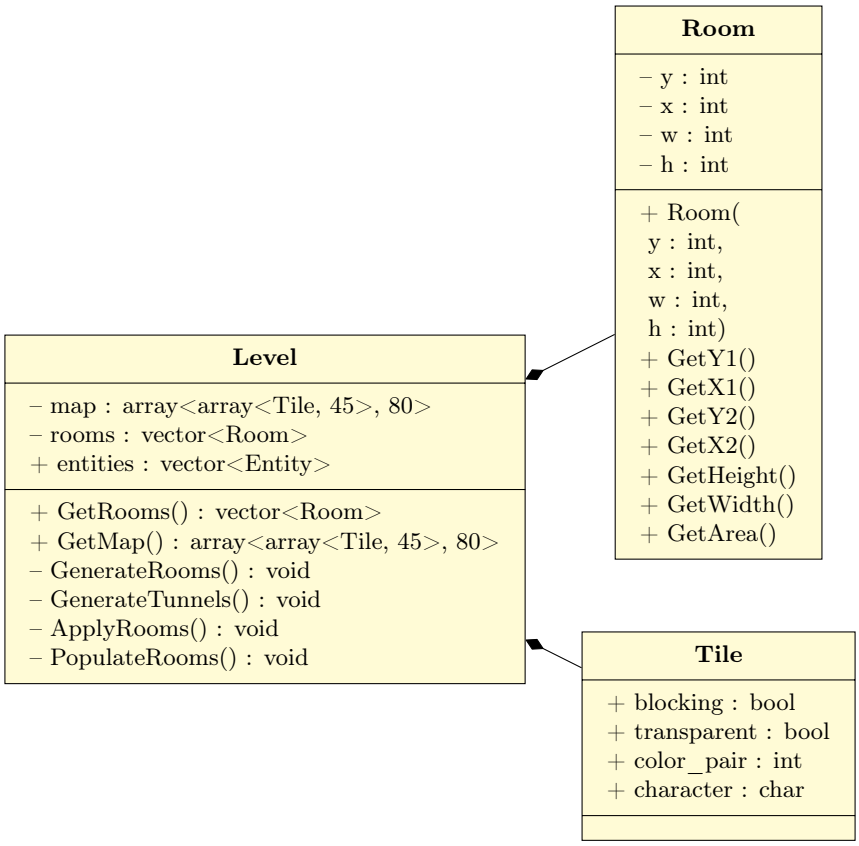

Figure 3: Level Class Diagram