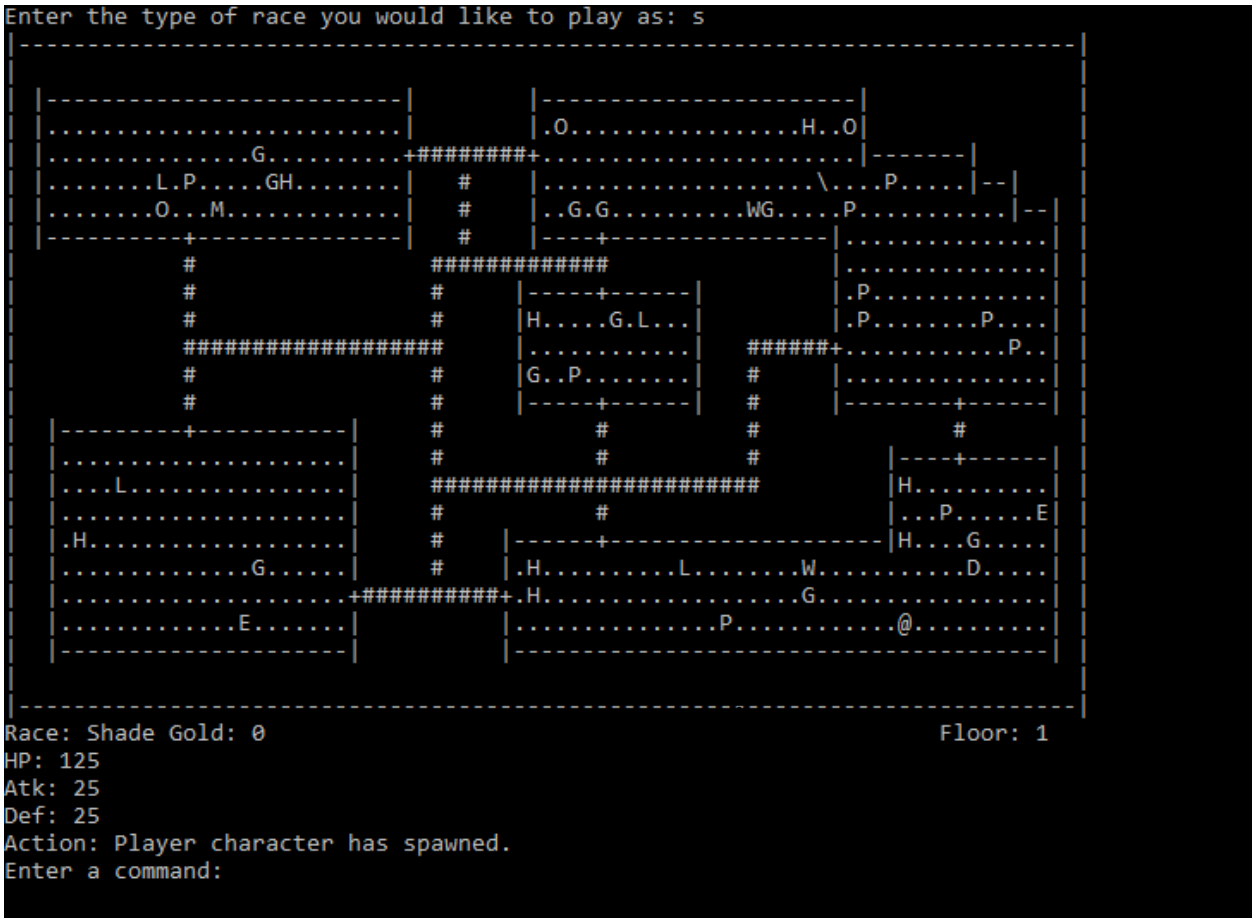
Design Document

ChamberCrawler3000
(CC3K)



Revision: 0.0.1
CS246 Final Project

Contents

Introduction

The design document will cover the different aspects of the ChamberCrawler3000's thought and implementation process. It will provide an overview of the design, components implemented for the game, and explain specific choices that we made along the way. ChamberCrawler3000 (CC3K) is a simplified rogue-like game, which redraws all elements every turn, rather than updating the window in real-time. It is a tile-based role-playing game where the player runs through dungeons, slaying enemies, collecting treasure and potions along the way until reaching the end of the dungeon on the 5th floor.

This current implementation of CC3K supports five character classes – shade, drow, vampire, troll and goblin – each with their unique abilities. In addition to this, this current implementation supports seven enemy types: human, dwarf, orcs, elf, merchant, dragon and halfling. With this are six different types of potions as well as types of gold, both of which spawn randomly around the map.

Overview

The project structure is divided into multiple sections, which all combine to create the final product. We created a superclass called "Race" that each specific race inherits from, as well as a superclass called "Enemy," where each type of enemy inherits from. These superclasses were created to promote code reusability and curb code redundancy as in both cases, many of the types of races and enemies share fields and methods. They are able to share fields and methods due to the fact that they are all a race or an enemy rather than their own individual types. Although they differ in their abilities, they share the same features. Therefore, defining these types of functions in their own superclass aids with avoiding code redundancy as each subclass does not have the same methods with the exact same definition. "Enemy" and "Race" are subclasses of the "Character" superclass, which holds fields that both enemies and races have, such as health, attack and defence. The character superclass also contains an isDead() method as this method wouldn't change whether the function is being called on a race or an enemy. We decided to implement our code this way to promote high cohesion and low coupling. This implementation allows for easy additions by simply creating a new subclass and inheriting from the superclass depending on whether the new class is a race or an enemy.

Similar to the race and enemy classes, we decided to create a superclass for the potions with effects, called "Effects," which holds the fields that all the potions share regardless of their personalized features. This superclass was created to avoid code repetition and increase high cohesion. As well, we made a base class that represents a potion with no effects to allow us to

use the Decorator design pattern. The potion superclass has the effects and base classes inheriting from it, which holds the functions used for all potions, and inherits from the “GameObject” class. Finally, the gold treasure has its own class, which differs from the potion superclass, and it holds the functions that are necessary for its implementation.

We decided to make every object inherit from an abstract “GameObject” class. We did this as we wanted to monitor all the objects in a 2D array in the board class. Making each class inherit from the GameObject class makes all of them a GameObject too and, therefore, can be held in arrays. The board class also contains an array that holds the five chambers. This was to help with the random generating of the items and enemies onto the map.

Schedule & Tasks

The schedule for the development of the project is shown below:

Task	DD1	DD2	DD3	DD4	DD5
	29/11/2021	01/12/2021	03/12/2021	07/12/2021	09/12/2021
.h Files					
spawn function					
nextTurn function					
board constructor					
getObject function					
setObjects functions					
chamber class					
race class					
shade subclasses(no attack functions)					
shade attack functions					
enemy class					
human subclasses no attack functions)					
human attack functions					
doorway class					
stairway class					
wall class					
showBoard					
Main.cc					
merchant class					
dragon class					
character class					
drow class					
vampire class					
hafling class					
troll class					
orc class					
dwarf class					
goblin class					
potion class					
gold class					
potion functions					
dragonHoard					

All: purple
Aryan: blue
Sherry: orange
Adi: green

Design

We designed the overall program such that every class, in addition to other things, extends from the GameObject class. The GameObject class behaves as an abstract class, with its purpose being to aid in the management of the different objects on the board. An array holding GameObject pointers is declared in the board class and is defined in its constructor as well as altered when functions such as move, spawn, reset are called. The array holding pointers prevents any object slicing or loss of data and is therefore secure. This method provides us with an efficient and simplistic method to monitor all the objects on the board, especially when the object's specification isn't required. An example of this is move, where the enemy class uses the getObject method in the board class to check for available spots for the enemy to move into. As well, in order to simplify spawning and moving in between chambers, we created a chamber class. The board class contains 5 of these chambers, and each chamber contains a reference_wrapper to the corresponding variables in the board class. This was done so that if we changed a space in the chamber class, the board class would also change, and vice versa. As well, this enabled us to move between chambers by having instances of a passage class in places where there are passages.

In addition, much inheritance was used regarding the "Race" class as well as the "Enemy" class. Both these classes are subclasses of a "Character" superclass, inheriting the fields and methods they share and would require. This curbs redundancy of the code as well as encourages high cohesion and low coupling. We continued by creating a subclass for each race and enemy type that was inherited from the Race or Enemy superclass, respectively. This implementation method allows there to be quick and easy additions due to low coupling. For example, if a new race type needs to be added, it can be defined the way it needs to be and then extend off the Race superclass. The same has been done in the enemy class. With this, we used virtual methods and overloading for each race/enemy subclass, allowing the methods to be specific according to the unique qualities each type possesses depending on what the parameters are. Finally, we used the visitor design pattern in both directions for the attack functions as the main player would be attacking the enemies and vice versa. For example, the attack function called for a vampire attacking a dwarf will have different results in comparison to the attack function being called for a vampire attacking a human. This was solved via method overriding. The two-way visitor design pattern promotes high cohesion and low coupling, allowing us to easily add new operations when adding new enemies.

What's more, we used the decorator design pattern for the potions class and all its subclasses. Like this, calling the calcAtk, calcDef, or calcHp methods can allow the program to easily traverse through all the potions within effect on the player, calling the same function on each potion and returning the overall ending effect after running through the entire list. The use of the decorator pattern allows greater flexibility in comparison to a static inheritance. In addition,

implementing and adding features simplifies through this method as it becomes more modular with low coupling.

Resilience to Change

When designing our game, we made sure that it had the ability to support various changes to the program specification. To begin with, one can easily add more potions to the program by simply creating another class that inherits from the “Effects” superclass. Since the “Potion” superclass holds the functions that all potions share, and the base class and the effects class inherit from it, it is sufficient to add a class under the effects class that has the specific attributes that the new potion has. In addition, the Effects superclass has a boolean variable that determines if the effect is permanent, hence it would be possible to add a potion that permanently increases attack or defense. If a potion was removed from the game, a subclass of Effects would be removed, however, no other class gets affected since each class holds their own features.

Next, adding in a type of race would just require adding in a class that inherits from the race superclass, and adding minor changes to the enemy classes inheriting from the enemy superclass. As well, to remove a race, one would simply need to delete a race subclass and the attack functions in charge of attacking that race from the enemy subclasses. The new race class would need to have its own constructor and all the attack functions that the other race classes have. As well, we would need to add an attack function in each of the enemy classes that will take care of attacking the new race. This is a simple addition to the program as the race superclass contains the functions that all types of races require. The process would be similar for adding or removing a new Enemy class.

Answers to Questions

How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races?

We implemented the various races by creating a race superclass with five subclasses each of which are a specific type of race, inheriting from the race class. This form of inheritance reduced the redundancy of code as functions that all races would carry out in the same way did not have to be redefined multiple times for each subclass. Instead, all the subclasses inherited it as it was defined in the race superclass. For example, functions such as move(x, y, b) or usePotion(potion) are functions that all races would require to use. These functions are not specific to a race and therefore, can be defined in the race superclass while the subclasses inherit the method. Furthermore, this method of implementation highlights high cohesion and low coupling. This is why adding another race wouldn't cause any errors. A new subclass of race can simply be extended from the race superclass with all its attack functions.

How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

We implemented the enemies class in a similar way to how the characters and races classes were produced. There is an enemy superclass, while each specific type of enemy extends from the superclass, having its own specific characteristics that are different from the other enemies. This reduces the redundancy of code for methods, as methods such as the move function that all enemies regardless of type would have to do, this function would not need to be defined in each specific class. Furthermore, countless if statements and overloaded functions would be required if not having this implementation. However, we can send a reference to an enemy object as a parameter and no object slicing would occur. We can see from the implementation that both the race and enemy classes are inheriting from a larger superclass called "Character". This allows fields such as HP, atk etc, to not be redefined in the classes.

How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

In order to implement the various abilities for the enemy characters as well as the player characters, we will be using a Visitor pattern that goes both ways. That is, when attacking, each type of enemy character can visit each type of player character, and each type of player character can also visit each type of enemy character. The reason we decided to use a Visitor design pattern is because a "battle" between one type of player and one type of enemy can differ between a battle between a different combination of player and enemy types (e.g. orcs do 50% more

damage to goblins). Thus, the Visitor pattern's ability to perform double dispatch allows the attack to depend on the player's type and the enemy's type. We have a Visitor pattern going both ways because a player can attack an enemy, but an enemy can also attack a player.

The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

We think that overall, the Decorator pattern would work better when modelling the effects of potions. The main reason for this is that multiple potions can be in effect at a time, and so with the Decorator pattern, we only have to create one class for each type of potion. If we were to use the Strategy pattern, we would likely have to create a class for each combination of potions. As well, if we were to add a new type of potion with the Decorator, we would only need to create one new class, but with the Strategy pattern, we would have to create multiple. Additionally, with the Decorator pattern, it is easier to work with potions that have a permanent effect. If a potion has a permanent effect (indicated by the permanent field), then its effects will be applied to the player, and it won't be added to the linked list containing all the potions in effect, whereas a non-permanent potion would be added. However, an advantage to using the Strategy design pattern is that the class structure would be simpler, and we wouldn't have to worry about deleting heap-allocated memory. Regardless, we think the Decorator design pattern is better overall.

How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

In order to ensure that the generation of Treasure and Potions reuses as much code as possible, we could create a function that takes in an integer and an array with n doubles. The ith element in the array would contain the probability of generating the ith item, and the function would return an integer between 1 and n (inclusive), representing what item should be generated. This function would be used when generating both Treasure and Potions in order to determine what type of Treasure or Potion to generate. For example, when generating Treasure, an array with three elements would be passed in. If the function returned two, we would spawn the corresponding pile of gold (e.g. small gold). Similarly, when generating Potions, we would pass in an array with six elements.

Extra Credit Features

Smart Pointers

In our code implementation, we used shared pointers, which are smart pointers that allow multiple pointers to point to the same object. We have decided to use shared pointers throughout the whole project rather than using new, to increase efficiency in our code, as that eliminates the need for deleting heap-allocated memory. In addition, we decided to use shared pointers rather than unique pointers as each object has multiple owners to it (e.g. a player is owned by the board class, the game variable in board, and the chamber class). However, this presented us with the challenge of cyclic references in our code, which would not be the case had we used new to allocate memory. Therefore, when writing the shared pointers, we had to be careful to avoid cyclic references, as this would result in memory leaks.

Read in Enemies by Command

We implemented the ability to read in enemies from the input file rather than spawning them when a file is given. With that feature came the challenge of reading in a dragon and figuring out a way to know that the dragon is associated with a particular hoard. This problem arose because the constructor for the Dragon class takes in the coordinates of the hoard it is guarding. To solve that, we created a vector that stored and kept track of the coordinates of the hoards that don't have an associated dragon yet and dragons that don't have an associated hoard yet. Then, when we read in a dragon or a hoard, we checked if the vector contained coordinates of a dragon or a hoard, depending on what is read in, that is right beside it. If suitable coordinates are found in the vector, then we make a dragon object and remove those coordinates from the vector. Otherwise, we push the coordinates of the current object into the vector.

Final Questions

What lessons did this project teach you about developing software in teams?

This is the first time each of us has worked on a development project in a team setting and we can all agree that we have learned a lot. To begin with, it is important to strictly divide up the project into different components and give each member their own responsibility. During the planning process we split the work as evenly as possible between the three of us, and set due dates that we were all comfortable with to complete each part. Next, it is important to keep

everyone in the loop when creating major changes to the code. This is true for multiple purposes, firstly, major changes in one part of the program can often affect other parts of it, hence members need to be aware and adapt their work accordingly. In addition, by talking it out we were able to simplify complicated implementations that were not thought of during the planning process. Next, we learned that it is important to trust your group members when developing software in teams. As we have never worked together, it was hard for us to trust each other to code, we would often have doubts which caused some tension. However, we quickly realized that we all have the same end goal, and got over this hump which allowed for a smooth process afterwards. Lastly, we learned the importance of writing clean, and readable code. When debugging, we found it difficult to understand the other person's code sometimes. Although we were always available to answer any questions, more time would have been saved if we all understood each other's code implementation right away.

What would you have done differently if you had the chance to start over?

If we had the chance to start over, we would do a couple of things differently. Firstly, we would compile the code after each due date that we set for ourselves rather than waiting for the end to compile. We realize that by compiling at the very end, we had many problems that occurred in different places that could have been avoided or fixed prior if we had compiled earlier. Next, we would have looked more into the major implementation components and used that to estimate the time it would take to complete them. We found that we underestimated the amount of time that it would take to implement the board class and attack functions, which created stress for the group members in charge of implementing those parts. Lastly, we should have figured out the implementation of the chambers earlier to avoid spending a lot of time on adjusting it later. We originally planned to use a vector of references that contained references to the corresponding variables in the board class. However, we later realized that a vector of references is not possible because references cannot be bound to another object once initialized. So instead, we used the reference_wrapper class and had to change every chamber-related function to use reference_wrapper instead of references.

Conclusion

Overall we can all agree that we immensely enjoyed working on this project. This experience has taught us many lessons that we will all take with us and implement for future group projects and workplaces. The last three weeks have been very stressful but also very rewarding. We are very proud of what we have created and hope you will enjoy marking it!