Project Plan Final - cc3k

Authors: Angelina Liang (a25liang), Derek Tan (d28tan), Paul Xiao (pxiao)

Overview

The game will first process the command line arguments, initialize input/output (I/O) resources, and enable the corresponding features based on the inputs.

Then, it would give the game engine `CC3K` access to those resources.

The game engine will create a menu for the player to select their desired race, then pass in the information to initialize a new game instance. Meanwhile, the instance of the menu is simultaneously destroyed.

The game will construct a new player based on the information given and initialize all in-game flags (e.g. `hostile_merchants`).

Then, it would start level generation per level, starting with the first level.

When the player goes up/downstairs, it will fetch the correct level from its stored list of levels to run. This design is to accommodate add-ons related to revisiting levels (floors).

The game will primarily run based on user input. It will pass the input command to the player, if the player takes a legitimate action, (i.e. the command is valid) then it will let enemies take their respective turns (unless the command for stopping movement has been issued).

The `level` class will contain all the information of that level (i.e. map, items, enemies) and is responsible for providing the related details (available locations, pointers to items/enemies) for the game to process.

The `map` class stores all map-related information (room numbers, walls, corridors, etc.) and handles random map generation.

If the player picks up any items (e.g. potions and treasures), the item ownership will be transferred over to the player class.

When a potion is applied to a character, the potion will be moved into the “effects” list of the character. The related stats of a character will get reset before each turn, and the corresponding potion effects will be applied to them to deduce the temporary stats for that turn.

This design is meant to allow players to apply potion effects onto enemies (via throwing potions) and to accommodate the turn-based duration of some additional DLC potions.

For attacks, the attacker will call the `get_hit()` method of the character being attacked, and both functions can be overridden by specific subclasses to implement custom special abilities.

Command Line Arguments

Optionally, the `cc3k` program can be run with additional command line arguments:

You can run `./cc3k -h` or `./cc3k --help` to open the help information in the terminal.

Argument	Description
`-n`	Uses ncurses for I/O
`-r`	Randomly generate maps. Cannot be used with `-m`
`-c`	Enemies chase the player
`-d`	Enemies can go through doors
`-i`	Enable inventory (will also enable `-o`)
`-t`	Enable throw (will also enable `-i`)
`-R`	Enable revisiting levels. Cannot be used with `-m`
`-e`	Enable extra potions and races
`-E`	Enable extra levels. Cannot be used with `-m`
`-o`	Allow players to pick up gold and potions by walking over them
`-m [file]`	Reads map data from a file. Cannot be used with `-r` or `-R` or `-E`
`-s [seed]`	Sets initial seed
`-I [file]`	Reads commands from a file. Cannot be used with `-n`
`-O [file]`	Outputs to file. Cannot be used with `-n`
`-h/--help`	Displays options list (doesn't start a game)
`--races`	Displays playable characters list
`--enemies`	Displays enemies list
`--potions`	Displays potions list
`--commands`	Displays available commands

Design

The following paragraphs describe the techniques used to solve the various design challenges encountered in this project.

For player character (PC) and enemy classes, we added two virtual classes: `player_base` and `enemy_base` that were implemented with a default (generic) set of interactions. To implement the logic for each specific PC/enemy class, a subclass would inherit from the `player_base` or `enemy_base` class and override the logic to match the respective characteristics or abilities of each character.

Gold and potions both inherit from a common `item` class. Potions are implemented with a decorator design pattern, such that their effects can be immediately applied by decorating the player class.

Abstract I/O classes are used to provide easy access to different I/O methods (`cin` / `cout` , file I/O, ncurses I/O).

At the start of each turn, the player stats are reset to accommodate the turn-based duration of potions. In addition, a timer is used to keep track of level-based duration, potions that last beyond the scope of a particular level.

For output, the output is not redisplayed upon every state change. Instead, it is written to a buffer and rendered all at once when the `out::render()` method is called.

If provided with a file for the five floors, when initializing the game (a CC3K object), it will preprocess all the given data to store information about the entities on the map and use a DFS algorithm to number the tiles for each room.

Random Map Generation is handled in the following manner:

- Generate the dimensions of all rooms
- Distribute the rooms horizontally into layers
- Distribute the layers vertically
- Jitter (randomly move upwards or downwards) the rooms vertically
- Generate the doors for each room
- Draw passages connecting the doors for all rooms

Ensuring that the player and stairwell do not spawn in the same room requires a way to identify the different rooms. Therefore, tiles in rooms are stored as numerical digits instead of '.' to keep track of the room number.

Resilience to Change

There are intentional design choices in the code structure that allow for the possibility of changes to the program specification.

If the developer wishes to add a new player character, enemy, potion, or I/O channel into the game, they can override the virtual functions in the class hierarchy to have easy access to the individual features at an abstract level.

There is a unified constants file (`src/constants.h`) that tracks the state of all constants used in the cc3k program. These include player stats, enemy stats, potion stats, feature flags, and map stats. The developer would only need to edit this file to make any changes to the stats used for the game.

There are a few utility libraries that are used across multiple different classes. These include: `rng` , `position` , and `fraction` . These tools have been abstracted away from the client classes to reduce the coupling of their common functionality and improve coherence among all game-related classes.

Answers to Questions

Question 1: 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?

Answer: In our project, two classes handle races: `player_base` and `enemy_base`. The abstract character and enemy classes will contain common attributes among all classes, such as attack (ATK), defense, (DEF), health points (HP), and hit rates.

Player character (PC) races (e.g. shade, drow, vampire, troll, goblin) will be subclasses of the class `player_base`.

Enemy character races (e.g. human, dwarf, elf, orc, merchant, dragon, halfling) will be subclasses of class `enemy_base`.

By design, it is manageable to add additional DLC classes to the game without refactoring the logic of the player character and enemy classes of the base game.

- If the developer desires to add a new PC class, they create a subclass of `player_base`.
- If the developer desires to add a new enemy class, they create a subclass of `enemy_base`.
- After creating the subclass, the developer may add any custom attributes/methods of the additional race into the new subclass (e.g. lifesteal, extra gold, etc.)

Question 2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

Answer: The system handles initializing different enemies to player characters in a similar manner. For enemy classes, the new enemy instance is inherited from the class `enemy_base` and the virtual functions of `enemy_base` are overridden in the implementation of each concrete enemy class.

The difference in generating between the player character and enemy classes is that enemy classes rely on a random number generator (RNG) object with predefined odds specified in the document, whereas the player character class generates with 100% certainty the selected player character class.

The enemy classes are designed in this way to make it easy to add new enemy classes in the future (as part of a DLC), all the developer would need to do is to inherit from `enemy_base` and add the custom functionality in its implementation file.

Question 3: How could you implement the various abilities of the enemy characters? Do you use the same techniques as for the player character races? Explain.

Answer: Enemies have a set of basic abilities (normal attacks, normal movement, getting hit by others) in their base class `enemy_base`, all of which are virtual and can be overridden.

Each basic ability in the `enemy_base` class also contains a default implementation, such that if a new enemy class does not change its default functionality, then the virtual function can simply be inherited and not overridden (i.e. non-pure virtual function).

These abilities are common amongst all enemies, and thus are shared through inheritance by `enemy_base`.

The same technique will be applied to the abilities of PC races (attack, applying potions, getting hit, etc.). For example, the basic set of operations for PCs is based on the shade race (i.e. shade doesn't need to override any method of `player_base` and is a generic PC class) and the vampire will override the attack function to include HP draining (i.e. lifesteal).

The `player_base` class also contains a set of default ability implementations that may be overridden or simply inherited as is, depending on the specific player character's race.

Question 4: What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

Answer: We would use the Decorator design pattern to temporarily attach the effects of temporary potions (WA/WD/BA/BD) to the PC class. Immediately upon potion use, the decorators will directly alter the ATK/DEF/HP attributes of the player, thus the potion does not need to be explicitly tracked for its effects to be reflected in the player's stats.

At the end of each level, there will be a reset function that will reset the effects of all temporary potions used on the current floor, thus clearing its effects before advancing to the next floor. After the level is complete, all player character stats will reset to their default values (except HP), and a new round of interpreting will begin at the new level.

Question 5: 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 the generation of treasure does not duplicate code?

Answer: Both potions and gold (treasure) will inherit from a base class `item`. The generation function will have shared functionality in generating positions, and different type indicators can be inserted to create items of various flavours (i.e. different potion types and different gold types).

Since potions and gold inherit from the base class `item`, any possible duplication of code is avoided by the sharing of common functionality (e.g. position generation). This makes it a more efficient process when adding new potion/gold types, as well as improving the developer experience when maintaining different items in the game (since core item functionality is centralized in a single location).

Extra Credit Features

No Raw Pointers

What we did: All memory in the program is allocated using STL containers and smart pointers, and no raw pointers are used to manually allocate memory.

`ncurses` I/O

What we did: Print output using the `ncurses` library, which reduces latency.

Random Map Generation

What we did: Randomly generates a new map each time.

Why it was challenging: Random generation can get out of hand since there is no way to predict the generated output.

How we solved it: Setting bounds on the parameters used for generation (max/min room height/width, room spacing, reserved space).

Enemy Chasing

What we did: Enemies move towards the player when within a certain radius.

Transient Enemies

What we did: Enemies can walk through doors/passages.

Inventory

What we did: The player can store potions and items in their inventory (across levels).

Why it was challenging: Managing potion ownership.

How we solved it: Levels can release the ownership of individual potions and transfer it to the player class.

Throwable Potions

What we did: The player can throw a potion in their inventory for up to a certain distance.

Additional Player Characters

What we did: Added new playable character classes to the game, each with their unique characteristics and abilities.

How we solved it: Override individual members of the `player_base` class.

PC Class	HP	ATK	DEF	Base Hit Rate	Description
T-800	800	40	50	2/3	All potions will give it rusty joints (i.e. -50 HP)

PC Class	HP	ATK	DEF	Base Hit Rate	Description
Mr. Goose	130	25	20	Infinity[1]/1	All potions are known at the beginning of the game
Monk	100[2]	70	0	1/1	Gains 2 HP at the start of each turn
Tavern Brawler	120	20	15	3/4	Has a 50% chance of attacking twice and a 50% chance of attacking three times
Assassin	100	30	10	1/1	Upon a successful hit, has a 10% chance of assassinating the target (instant kill)

[1] Mr. Goose is knowledgeable and will never miss an attack.
[2] Monk starts the game with 100 HP, with a max of 125 HP from potion effects.

Additional Enemies

What we did: Added new enemy classes to the game, each with their unique characteristics and abilities.

How we did it: Override individual members of the `enemy_base` class.

Enemy Class	Symbol	HP	ATK	DEF	Base Hit Rate	Description
Viking	`V`	150	30	25	1/3	Attacks twice vs. every race
Swordsman	`S`	100	25	15	1/1	Attacks with finesse (guaranteed hit)
Leprechaun	`l`	80	10	15	1/2	Steals 3 gold from PC on each successful attack If PC has insufficient gold, then deal damage with a much higher ATK Upon death, drop all stolen gold + 5 extra gold
Witch	`Z`	100	20	15	1/2	Upon a successful hit, has a 1/5 chance of applying a random potion onto PC (effect begins on the next turn)
Hacker	`h`	90	15	30	1/2	Outputs a random message when attacking / being attacked

Enemy Class	Symbol	HP	ATK	DEF	Base Hit Rate	Description
Baby Dragon	`B`	140	20	40	1/3	Not fully grown, thus able to move and is immune to all potions

Additional Potions

What we did: Added additional potion classes that had a variety of different effects not included in the base game.

Why it was challenging: Additional potions have turn-based duration instead of lasting for an entire level.

How we solved it: All potions have an internal duration timer and we explicitly stored potions in the ownership of the character owning the effect.

Potion Class	HP	ATK	DEF	Hit Rate	Description
Continuous Restoration (CR)	+3	-	-	-	Lasts 5 turns
Savage Strike (SS)	-	1.25x base	-	0.8x base	Lasts 20 turns
Echoing Resilience (ER)	+7	-10 base	-10 base	-	Lasts 20 turns
Tempest Tantrum (TT)	-25% current HP	3x final	0.5x final	-	Lasts 12 turns
Berzerk Brew (BB)	-	2x base	0.5x base	-	Lasts 15 turns
Borrow Life (BL)[1]	+50 start, -55 end	-	-	-	Lasts 24 turns
Fine Booze (FB)	+2 per turn	-	-	0.7x final	Tavern brawlers never miss Lasts 12 turns
Ironclad Ward (IW)	-	0.5x final	3x final	0.75x final	Lasts 12 turns

[1] Overhealth (additional health above max race HP) gained by Borrow Life cannot be restored by other potions.

Final Questions

Q1: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

A: Many important lessons were learned throughout the course of doing this project. Specifically, it reminded us of the importance of version control, project/task tracking, and communication.

Importance of Version Control:

When simultaneously collaborating on a project with multiple contributors, we discovered that it is important to keep a single, centralized source of truth available at all times. Without it, there would be unnecessary confusion surrounding which version of the code was the latest, which version is the most recent working (i.e. compiling) version, etc.

For our group, using Git allowed us to have a stable working version of the code in the `master` branch at all times, as well as roll back any breaking changes to a previous working version of the code. Furthermore, it helped us keep track of the latest changes to the code using Git logs and commit messages and provided us with the ability to asynchronously work on different features without interfering with the master branch (through Git branching) or with each other.

Importance of Project and Task Tracking:

We also realized the importance of having a system to help us keep track of the current status of the project and the status of all pending tasks. This required a project management tool such as Trello to monitor and track the progress of such tasks.

Through a Kanban Board on Trello, we tracked the status of all tasks, bugs, chores, and features. By bucketing them into the categories of To-do, Doing, Blocked, Under Review, and Done. Furthermore, we could assign tasks to individual members to clarify the work distribution.

Importance of Communication:

Through this process, we realized that communication was an essential pillar of effective group collaboration. In our group, we maintained a searchable chat log that we would use to communicate, and this helped us organize our thoughts and ideas. Whenever the code or design was updated, a message was sent to the group notifying everyone of the changes in detail. This practice helped our group avoid potential miscommunications, as well as build a traceable knowledge base for future reference. We also constantly updated the Trello board and informed group members of any blockers/updates when they occurred. Through these processes, we were able to constantly stay on top of tasks and always maintained a clear understanding of what needed to be done.

Q2: What would you have done differently if you had the chance to start over?

A:

1. **More time spent on planning and design:** Many of our features were inspired while working on the project, and we overlooked a few requirements at the beginning.
2. **Unify coding style:** While working, we found that individual members have different styles for indentations, naming, code structure, etc. Even though we managed to present a unified-

looking codebase with the help of tools like AStyle, it required a lot of additional communication to help understand each other's code.

3. **Continuous testing:** We started out implementing individual modules and only writing testing harnesses specific to the modules. When we pieced them together after individual testing, we found that they did not coherently fit together. Moving forward, we should test both individual components (unit testing) as well as their interactions with other components (integration testing).
4. **Emphasis on documentation:** We had several loose files describing how each module and the entirety of the code worked but lacked any general documentation to explain the code in its entirety. We should document what we were doing with each piece of code along with how we did it.
5. **More robust structure:** We designed the code with some of the extra features in mind, but the features we implemented after required us to implement additional interfaces to accommodate. In the future, we should design interfaces with accessibility and modularity in mind.

Conclusion

As a concluding note, we would like to express our heartfelt gratitude to the CS246 instructors Caroline Kierstead and Jens Schmitz, as well as the CS246 instructional support staff, for their invaluable guidance, insightful advice, and unwavering support throughout this process.

Many DLC features in this project were inspired by NetHack, an open-source, single-player roguelike video game also inspired by the 1980 video game Rogue.

This project has been a rewarding and enjoyable experience, through which we gained substantial knowledge in C++ and object-oriented programming. The skills and understanding we acquired in this project will undoubtedly serve us well in our future programming endeavours.

After careful consideration, we have decided not to become Shades[1] and instead transform our lives into those of Hackers[2].

This project has received great help from Mr. Goose[3] during debugging, which is a good alternative to trolling[4].

This message was sponsored by your local neighbourhood leprechaun[5].

Sincerely,

Angelina, Derek, and Paul

[1] "You either die a hero or live long enough to see yourself become the villain" - Harvey Dent, in the film The Dark Knight

[2] Hackers will attack Shades in the cc3k game.

[3] The one who knows all.

[4] Troll was the most broken character in the basic game, thus Mr. Goose is a good alternative.

[5] They probably stole all your gold when you weren't looking.