CS 162: Assignment 4 Design Document

Crystal Lee

1 The Problem

This time we're making a game called "Escape from CS 162" which is a maze game. The player's objective is to make it to the immobile Instructor with enough Programming Skills (3) to pass. If they encounter the Instructor without enough skills, they fail the class and thus lose the game.

The player starts at a random place in the maze, and has to deal with TAs that roam the maze. If the student finds a TA (they are in the same or adjacent squares), then the player would be caught, lose all of their Programming Skills, and ultimately lose the game. The only way to deal with TAs are either to avoid them or to appease them by demonstrating a Programming Skill. Appeasement lasts for 10 turns.

Programming Skills are pickups that are randomly dispersed throughout the maze, but are guaranteed not to spawn where the Instructor is. They also don't stack (only one skill per square).

The maze is of a size given by the user when they boot up the game. The minimum is a 5x5 but there is no specified maximum. When the maze is a 5x5, there should be four walls total. It is unspecified if there should *always* be four walls, even for bigger board sizes.

There are no file inputs or outputs this time, just inputs and outputs to terminal. The program gets inputs in the form of moves (characters) from the user and outputs the current board.

1.1 Extra Credit

There are two chances for extra credit, which require extra implementation. The first is implementing an AI that plays for you. Though it doesn't seem to need to be a true AI, it still has to have a decent algorithm that attempts to win the game. It should play as the player does, only knowing about the current board (current player, skill, and TA positions). It says nothing about how good the AI's moves are, but perhaps it's better that way...

The other chance is to implement WASD controls such that we don't have to follow every command with an `Enter` key after it. It would make it play much more like a game and less like a program we wrote for class. Essentially, bonus points for smooth gameplay.

2 Proposed Solution

On a super high level, my implementation has a couple screens: the start screen (with board size), settings screen, and game screen. Each screen is its own object and controls both input from the user and output to the screen. Setup is done by the main function, though, to ensure that the screen isn't initialized multiple times. Below are some notes I made while thinking about the design. It's mostly UI.

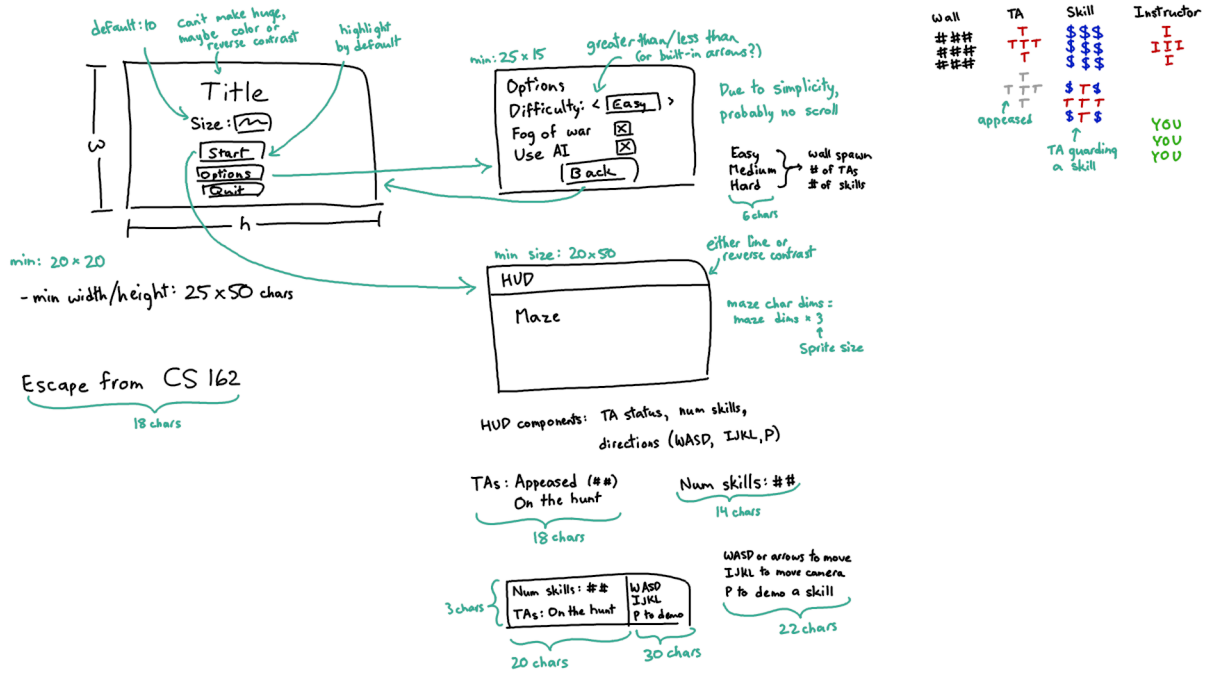

Figure 1: Notes for possible UI design.

Another note about implementation: there are no animations to handle. That means that everything is driven by user input and the next "frame" only changes when the user does something. This means the program doesn't need to deal with handlers for timers or user inputs to change animations or having to worry about frame rate anything like that. The only thing that requires handlers is for screen resizing (from the ncurses library).

2.1 Main Function

Arguably the most important function of the entire program: the main function, since it (directly or indirectly) calls everything else from within it. The header is standard, so just the pseudocode is below. The games runs on "screens" which are UI elements that point to each other.

```
initialize window for terminal
make a new GameScreen pointer called currScreen and initialize it to a new
    StartScreen

while currScreen isn't a nullptr (if it is, the user wanted to exit)
    run currScreen's input loop and store the resulting screen in nextScreen
    delete currScreen
    set currScreen to nextScreen
delete the window
```

2.2 Required Classes

The classes listed below are all required classes, often with some function or implementation requirements as well.

- `MazeLocation` (abstract class), which contains `public bool is_occupiable()` and `public char get_display_character()`
- `Wall` (extends `MazeLocation`), which always uses # as a display character
- `OpenSpace` (extends `MazeLocation`), which needs standard getters and setters for whether the space contains a student, TA, instructor, or skill
- `Maze`, which contains `public Maze(int, int)`, `public ~Maze()`, and `public MazeLocation* get_location(int, int)` and must contain vectors instead of normal dynamic arrays (why? For the specs.)
- `MazePerson` (abstract class), which contains `public char get_move()`, `public void set_location(int, int)`, `public int get_row()`, and `public int get_col()`
- `IntrepidStudent` (extends `MazePerson`)
- `TA` (extends `MazePerson`)
- `Instructor` (extends `MazePerson`)
- `Game`, which must contain a maze and the inhabitants of the maze (list of MazePerson objects)

2.3 GameScreen

We like top-down design, right? So I'm gonna start with the `GameScreen` class. This is a purely abstract class (some might even call it an *interface*) that defines some methods involving displaying things on screen. Each of them handles screen resizing and printing stuff to screen. The functions inside are listed below with a general description (none have pseudocode because they're all virtual).

```
// should be called when the window resizes so the screen can be updated
    accordingly
public virtual void* resizeHandler(int sig);

// input/output loop for this screen, run in main
public virtual GameScreen* loop();
```

2.4 Game

This is the main panel of the entire program, and extends GameScreen. It contains all of the maze related things (a Maze, vector of MazePerson objects) and has some extra fields regarding the state of the game.

Besides the normal HUD and maze layout, the screen managed by Game can have some alerts (winning or losing screens). These are managed by an internal state which shows whether an alert exists, which influences input (what keys are valid, what do they do) and output (draw maze or alerts).

The maze can be large. Larger than the terminal output, so some scrolling must be allowed to make sure the user can see all of it easily. This means that there needs to be some controls for it and functionality attached to it. Using IJKL, the user can do the thing.

Below is the pseudocode for the `loop()` function (defined in GameScreen), probably the most important of the functions in each GameScreen. We assume that the Intrepid-Student is first in the list of MazePerson objects, so that it appears that everything blocks until the user does something. This function requires the `std::istream.peek()` to work properly, which gets the next input char from cin without taking it out of the input buffer.

```
loop forever
    peek at user input and store the result in a new var (input)
    if we're in the main part of the game
        if the input isn't a move char
            consume the input
            do the camera or move things
            continue to the next loop
        run through every maze person and ask for its move
    otherwise we're in an alert
        consume the input and do what it needs (move the cursor, return, etc)
```

2.5 StartScreen

This is one of the two secondary panels, this one controlling the start screen. One of these is created on boot, and connects to the other two screens. Just like Game, this screen also has to keep its current state (cursor location, current maze size).

The `loop()` function pseudocode is below.

```
loop forever
    get user input and store the result in a new var (input)
    switch statement with possible user inputs (WASD, digits, space, enter)
        corresponding functions based on current state (select up/down,
    return another screen)
```

2.6 OptionsScreen

This is one of the two secondary panels, this one controlling the options screen. Just like Game, this screen also has to keep its current state (cursor location, current options).

On exiting loop, this should always get back to the start screen. It should also write all options to file (for reading from the Game class).

The `loop()` function pseudocode is below.

```
loop forever
    get user input and store the result in a new var (input)
    switch statement with possible user inputs (WASD, space, enter)
        corresponding functions based on current state (select up/down,
    return another screen)
```

2.7 File Format

To make this easier, the options screen outputs all settings to file. Each option is separated by spaces, and goes in this format: `difficulty fogOfWar aiUse`. Difficulty is represented by an integer (1 for easy, 2 for medium, 3 for hard) and the remaining two are represented by a 0 for false and 1 for true.

If the file is invalid or doesn't exist, the game should default to medium difficulty, no fog of war, and no AI use.

3 Test Coverage

The easiest way to test this is panel by panel, but all of them should error when the screen is too small.

3.1 Start Screen

Since there's a UI involved, instead of doing this in table format, here's a list of functionalities instead:

- right at boot, the cursor should start at the start button
- if all the user does is start the game, should default to a 10x10 maze
- clicking W or S should move the selection up or down, respectively, and wrap around
- clicking A, D, or any digit shouldn't do anything when on the start, options, or back button
- clicking A or D while on the maze size should decrement or increment the maze size, respectively
- clicking space or enter on the maze size shouldn't do anything
- clicking space or enter on the start button should boot up a new game of the specified maze size
- clicking space or enter on the options button should show the options screen
- clicking space or enter on the quit button should terminate the program

3.2 Options Screen

Once again, a list:

- when first opened, the cursor should start at the back button
- should have all settings from before (or default settings if nothing has been changed)
- clicking W or S should move the selection up or down, respectively, and wrap around
- clicking A or D shouldn't do anything when on the back button or checkbox options
- clicking A or D should make difficulty easier or harder, respectively, when on the difficulty selector
- clicking space or enter on the difficulty selector shouldn't do anything
- clicking space or enter on a checkbox option should select or deselect the option
- clicking space or enter on the back button should return to the start screen

3.3 Maze Screen

Final one. This one is the most difficult to test for, but here's a huge list for the main part anyway:

- when first opened, the camera should start centered on the player (or AI)
- should have the maze size specified on the start screen
- should use all settings from before (or default settings if nothing has been changed)
- fog of war makes it so that the player can't see any entities (MazePerson) except within a 5x5 square centered on them
- clicking WASD should move the player unless there is a wall or edge of maze (in which case, it skips their turn because they were stupid)
- clicking P should activate a programming skill
  - should appease all TAs and make their sprites grayed out
  - should see the TA status in the HUD say "Appeased" and the number of turns they're appeased for
- if appeased, the TA appeasement turns should count down for every move the user makes (like a usual status effect)
- clicking escape should bring up a quit menu (exit to start screen or exit entirely)
- if the user hits a TA and the TA isn't appeased, the lose screen should display
- if the user hits a TA and the TA is appeased, the user should be able to continue playing
- if the user hits the instructor and they don't have enough Skills, the lose screen should display
- if the user hits the instructor and they have enough Skills, the win screen should display
- clicking space or enter should skip the user's turn (they didn't want to move)

As for any alert dialogs that could pop up, they should disable all inputs to the actual maze portion. It should be WASD for moving the cursor around the options and space or enter for selecting.

Was that enough...? I wanted to start on this assignment ASAP... so I forwent a lot of the usual header plus pseudocode things so I could actually start coding.