Doki Doki CMSC Club

Po, Justin Andre E.

Tumulak, Patricia Lexa U.

Valles, Oscar Vian L.

Table of Contents

1 Introduction

Doki Doki CMSC Club is a role-playing experience that forces the player to make tough decisions and fight difficult battles for both romance and survival in college. "*Doki doki*" is a Japanese onomatopoeia for a heartbeat. Taking inspiration from both dating simulators and the turn-based battle system of Pokemon, the game immerses you into the life of a freshman student in P University or PU - the premier school of the country - starting their very first day. You get to engage in not only interaction with characters that will definitely make the player's heart go "*doki doki*" but also battle a variety of enemies to progress through the story. By choosing from a specific set of options, the player gets to follow their desired paths and reveal multiple different branching events that all play into the linear progression of the story. The game takes you on a journey through your first day in college - with things as mundane as introducing yourself and joining an organization, the university CMSC Club - and meeting and interacting with the different characters.

The game itself is structured to contain multiple paths, which means the choices the player makes may or may not have an affect on the story they are immersed in. These paths all converge into one point in the story, resulting in one ending - for now.

2 Characters

2.1 The Player

The main character of the game. An unnamed, ungendered student of P University, the premier university of the country, who is starting their first year. The game follows their story as they interact with the three other characters of the game and go through life as a college student.

2.2 Jeff Papadopolis

Jeff is one of the main love interests of the game, another student of PU who is in their fourth year and the CMSC Club President. He possesses a rather strong personality - coupled with his unfiltered mouth, he comes off as a very rude person. It isn’t obvious during the first impression but he’s also incredibly smart and at the top of his class - he isn’t the CMSC Club President for nothing. Despite his tough exterior, he is actually a fun loving guy who enjoys the occasional video game in his down time - when he isn’t studying. For some reason, he shows his softer side to the player at some point in the game.

2.3 Mr. K

Mr. K is another main love interest of the game, a young man in his early twenties fresh off grad school. He is the CMSC Club’s adviser and the player’s CMSC teacher. He’s a bit mysterious (but is just really lazy) and is really nice and genuinely cares about his students. He likes to read books of questionable material in his spare time (i.e. in the middle of class while the students do an activity). Generally friendly and likes to crack weird pun jokes. He doesn’t tell people (i.e. his students) his real name, just because he feels like it.

2.4 Chichi Santiago

A nice girl in the player’s class and a fellow member of the CMSC Club. She's perpetually shy and finds it hard to speak to new people and in front of crowds. She eventually becomes the player’s friend in the game.

3 Environment

3.1 General Description

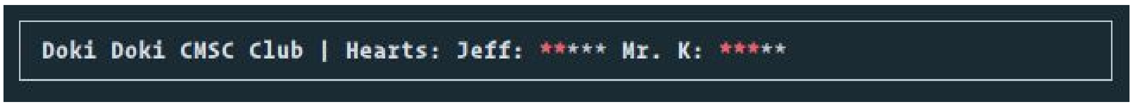
To implement the environment, ncurses was used. There are two main screens that are being shown to the users, the dialogue screen and the battle screen.

3.2 Dialogue Screen

3.2.1 Description

The user interface of the Dialogue Screen is comprised of three main components: The **Overall HUD**, **Dialogue Box** and the **Options Menu**.

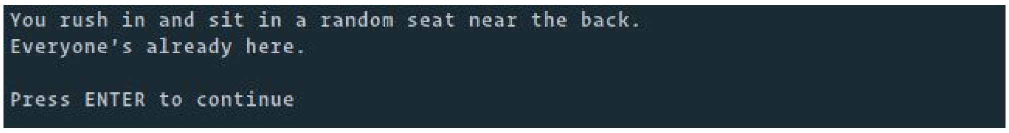
The *Overall HUD* component displays the name of the game, so that it will never be forgotten and the progress that you have with each character. This component updates at the start of each event. Events will be discussed in Section 4.



```
Doki Doki CMSC Club | Hearts: Jeff: ***** Mr. K: *****
```

Figure 3.1. Overall HUD Component

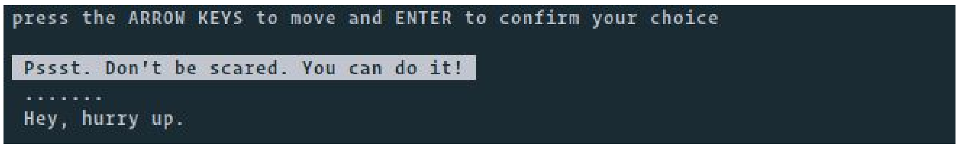
The *Dialogue Box* component displays the dialogue based on the event. The lines will be printed by groups, depending on the lines set in the event function.



```
You rush in and sit in a random seat near the back.
Everyone's already here.

Press ENTER to continue
```

Figure 3.2. Dialogue Box Component

The *Options Menu* displays the options that the user can select which are set by the event function.



```
press the ARROW KEYS to move and ENTER to confirm your choice

 Pssst. Don't be scared. You can do it!
 .......
 Hey, hurry up.
```

Figure 3.3. Options Menu Component

3.2.2 Implementation

The implementation of the Overall HUD Component, Dialogue Box and Options Menu can be found in hud.c lines 94 - 261. In line 229 - 261, the function called `createGameScreen()` will be called. This function will receive 5 arguments - `line`, `lines`, `option`, `options` and `incomingInfo`.

`line`, and `option` are an array of strings that contain the lines and options that the event needs. `options` and `lines` are the number of strings in the array.

`incomingInfo` is a struct defined by `gameInfo`. `gameInfo` is defined in gameInfo.h and it contains the following identifiers - `end`, `nextEvent`, `errorCode`, `interestPoints` and `hearts`. `end` is a flag that tells the main loop that the game has ended, `nextEvent` contains the event code for the next event. `errorCode` contains the errors that the game may have for easy debugging, `interestPoints` track your progress with a character and `hearts` is `interestPoints` divided by 100 which is used for the Overall HUD. This struct is passed around the various functions within the game to keep track of the progress that the character has.

`createGameScreen()` is a function that handles the windows that are created by ncurses and calls the functions `createHUD()`, `createContentScreen()` and `createOptions()` that create the components of the game screen. This function returns the choice selected by the user in the Options Menu which is created by the function `createOptions()`.

`createHUD()`, found in hud.c line 94 - 138, is a function that creates the Overall HUD component of the game screen. It takes the arguments `hudHeight` and `incomingInfo`. `hudHeight` contains the height of the hud. The function takes the height and width of the terminal, creates a window with the width of the terminal and the height as defined by the `hudHeight`, creates a box around it and prints the labels and the hearts through loops. `wattron()` is also used here to add color to the hearts.

`createContentScreen()`, found in hud.c line 140 - 174, is a function that creates the Dialogue Box component of the game screen. It takes the argument `contentHeight`, `starty`, `line` and `lines`. `contentHeight` is the height of the content, `starty` is the line where the window should start at, `line` is the array of strings that are passed from the event function and `lines` are the number of lines in the function. The function prints the lines in the window and prompts the user to press Enter to continue to view the next lines.

The full implementation of this can be found at section 6.2.2.1 as the Dialogue Box is also a part of the control scheme of the game.

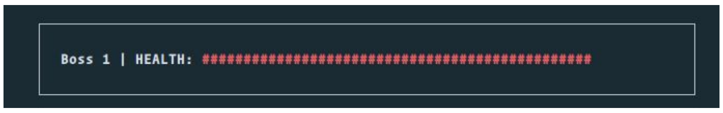
`createOptions()`, found in hud.c line 176 - 226, is a function that creates the Option Menu component of the game screen. It takes the `arguments`, `optionHeight`, `starty`, `option` and `options`. `optionHeight` is the height of the content, `starty` is the line where the window should start at, `option` is the array of strings that are passed from the event function and `options` are the number of options in the function. The function prints the options in the window and it enables the usage of the arrow keys to select which option the user chooses. After the user has selected enter, the choice is returned back to the `createGameScreen()` function. The full implementation of this can be found at section 6.2.2.2 as the Option Menu is also a part of the control scheme of the game.

3.3 Battle Screen

3.3.1 Description

The user interface for the Battle System is comprised of four main components: **Opponent Health**, **Status Box**, **Player Health**, and **Player Actions**.

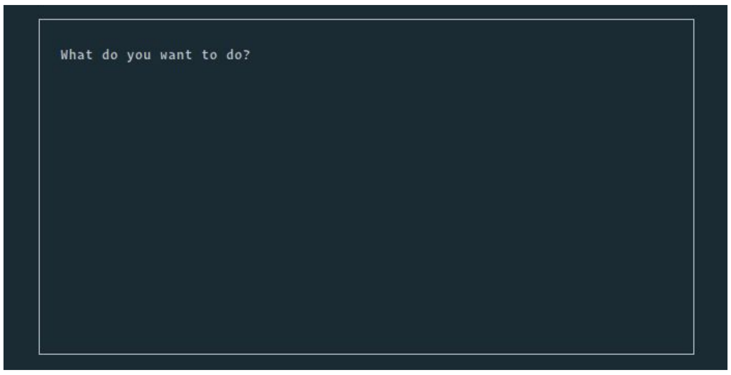
The *Opponent Health* component displays the name of the opponent and his corresponding hit points. This component continuously updates as the battle goes on, responding to the different actions of both the opponent and the player.



```
Boss 1 | HEALTH: ###############################################
```

Figure 3.1. Opponent Health Component

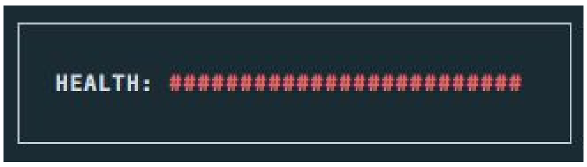
The *Status Box* component displays the outcome of the actions done by the opponent and the player. This component would display whether the opponent's action was successful or whether the player's action failed, displaying different outputs depending on the capabilities of both the player and the opponent.



```
What do you want to do?
```

Figure 3.2. Status Box Component

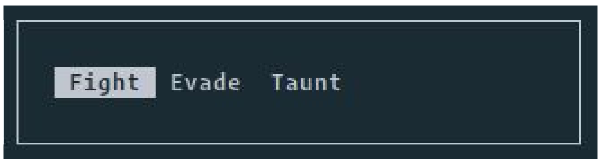
The *Player Health* component displays the health of the player. This component continuously updates as the battle goes on, responding to the different actions of both the opponent and the player.



```
HEALTH: ###########################
```

Figure 3.3. Player Health Component

The *Player Actions* component displays the different actions the player is capable of doing. The contents of this component would change according to the opponent faced.



```
Fight  Evade  Taunt
```

Figure 3.4. Player Actions Component

3.3.2 Implementation

The HUD for the Battle Screen is created from the battleSystem.c file. Each component is created or updated depending on the needs of the Battle Screen. It is created or updated through 4 functions - `createEnemyHud()`, `createContentHud()`, `createPlayerHud()`, `createOptionHud()`.

`createEnemyHud()`, found in hud.c line 263 - 287, takes the following arguments: `boss` and `hudHeight`. `boss` is a struct that contains the name, health and maximum health of the boss and hudHeight is a identifier that determines the height of the HUD. The function creates a window that has the width of the terminal and the height as defined by `hudHeight`. The function takes the length of the boss and other characters so that the remaining space can be calculated which is placed into the identifier `remainingSpace`. The health percentage of the enemy is calculated and is multiplied by `remainingSpace`, this value is then placed into the identifier `healthPercentage`. This value determines the number of characters to be printed when displaying the health bar of the enemy. The health bar is then printed through the use of loops.

`createContentHud()`, found in hud.c line 363 - 376, takes the following arguments: `hudHeight`, `line` and `lines`. `hudHeight` determines the height of the window, `line` is the array of strings to be printed and `lines` is the number of strings in the array. The function creates a window with a height determined by `hudHeight` and with a width the size of the terminal. The function then prints the array found in `line`.

`createPlayerHud()`, found in hud.c line 289 - 311, takes the following arguments: `player`, `hudHeight`. `player` is a struct that contains the health and the maximum health of the player. `hudHeight` defines the height the HUD. The same process is being done to calculate the number of characters to be printed for the health bar of the player as found in the function `createEnemyHud()`, the main difference is the omission of the player name and the width of the window, as it only takes half of the width of the screen.

`createOptionWindow()`, found in hud.c line 313 - 361, takes the following arguments: hudheight, `option` and `options`. `hudHeight` defines the height of the HUD, `option` contains the array of strings of options and `options` is the number of strings in the array. This function creates a window with the same height as the player hud and half the size of the screen. Options are dynamically printed on the same line and the user can select their choice using the arrow keys. The function will return the choice that the user has selected for use in the next events. As `createOptionWindow()` is part of the control scheme, it is fully explained in section 6.3.2.

4 Stages/Levels

4.1 General Description

The game’s concept of Stages/Levels comes in the form of events. There are two types of events in the game, dialogue events and battle events. These events are identified by their event codes, represented by an int and these event codes tell the system where the function of each event is located in the event switcher.

4.2 Dialogue Events

4.2.1 Description

Dialogue Events contain the exposition, dialogue, options and the results of these options. The majority of the events in the games are dialogue events.

4.2.2 Implementation

The implementation of Dialogue Events can be found in gameEvents.c. Each event has a function that contains the lines and options that the use will see. It also includes the result of these options which can affect the interest points, next event branch and the overall progress in the story. The events use the struct `gameInfo`, which was explained in section 3.2.2, to pass information from one event to another. This includes `nextEvent`, `interestPoints`, end and `errorCode`. This enables the system to properly keep track of the information throughout the duration of the game. The following code is an example of a dialogue event:

```
gameInfo onePathTwo(gameInfo _eventInfo) {
   int lines = 9;
   const char *line[lines];
   int choice = 0;

   line[0] = "The girl slowly gets up from her seat and speaks in a quiet
voice.";
   line[1] = "";
   line[2] = "Girl: Umm... My.. name is.. Chi-chi Santiago. It'snicetomeetyou!";
   line[3] = "";
   line[4] = "With that, she falls back into her seat.";
```

```
    line[5] = "She dips her head to hide behind her hair.";
    line[6] = "";
    line[7] = "'Wow she's really shy...'";
    line[8] = "";

    //Setting Options
    int options = 2;
    const char *option[options];
    option[0] = "Ignore her.";
    option[1] = ".......";

    createGameScreen(line, lines, option, options, _eventInfo);

    switch (choice) {
        case 0:
            _eventInfo.nextEvent = 101;
            _eventInfo.end = 0;
            break;
        case 1:
            _eventInfo.nextEvent = 101;
            _eventInfo.end = 0;
            break;
        default:
            _eventInfo.errorCode = 2;
            _eventInfo.end = 1;
            break;
    }

    return _eventInfo;
}
```

Figure 4.1. An example of a Dialogue Event

Each function that is a dialogue event contains the following identifiers: `line`, `lines`, `option`, `options` and `choice`. `line` is an array of strings that contain the lines that the user must view in the event. `lines` contain the number of strings in `line`. `option` is an array of strings that contain the options that the users can select and `options` is the number of strings in the array.

The function then takes the identifiers and passes it to the `createGameScreen()` function which displays the game screen as explained in section 3.2.2. After the `createGameScreen()` function returns the choice that the user has created, a switch statement is used to set the correct values on the necessary identifiers in the struct. The function then returns the updated struct.

4.3 Battle Events

4.3.1 Description

Battle Events are the events that handle the battles and the effects that the battle has on the overall progress of the story.

4.3.2 Implementation

The implementation of Battle Events can be found in gameEvents.c. The following code is an example of a Battle Event function:

```c
gameInfo battleFour(gameInfo _eventInfo) {
   int result = bossBattle(4, _eventInfo);

   switch (result) {
       case 0:
           _eventInfo.nextEvent = 1204;
           _eventInfo.interestPoints[1] += 50;
           _eventInfo.end = 0;
           break;
       case 1:
           _eventInfo.nextEvent = 1000;
           _eventInfo.interestPoints[1] += 75;
           _eventInfo.end = 1;
           break;
       default:
           _eventInfo.errorCode = 2;
           _eventInfo.end = 1;
           break;
   }

   return _eventInfo;
}
```

Figure 4.2. An example of a Battle Event

Each battle event calls the `bossBattle()` function, which will be discussed in section 5.3. The function will pass on the boss code and the gameInfo struct to the `bossBattle()` function. After the battle, the `bossBattle()` function will return the result of the battle. Corresponding effects on the overall progress of the story will be set using switch statements and modifying the corresponding identifier on the gameInfo struct. The updated values will be returned by the Battle Event function.

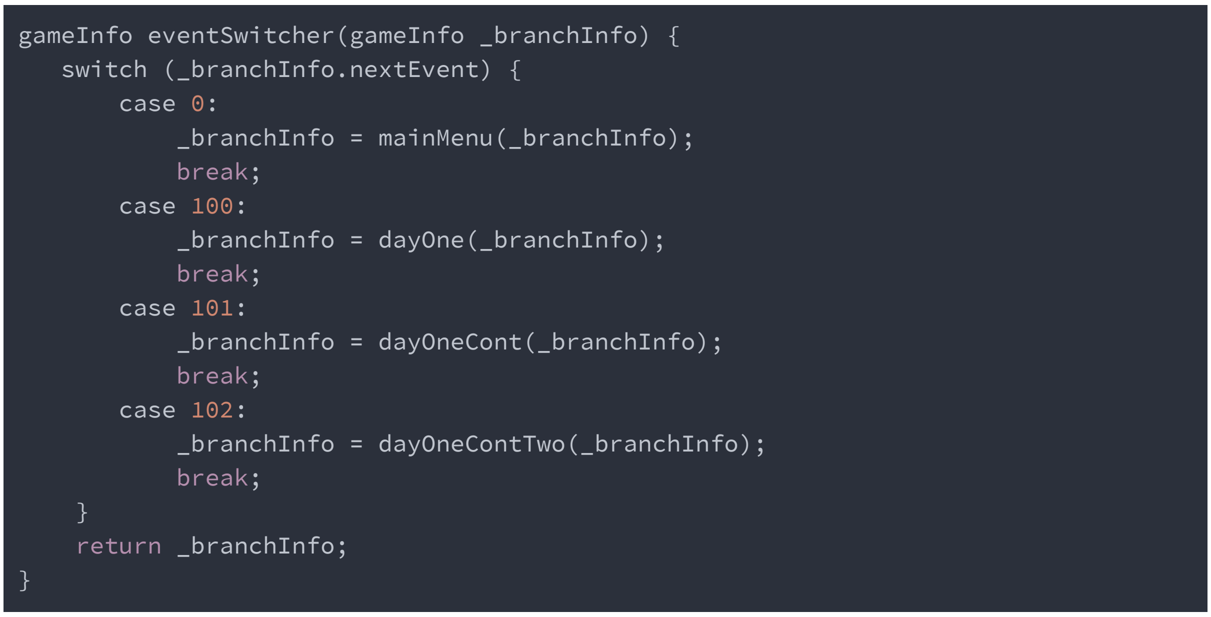
4.4 Event Switcher

4.4.1 Description

The Event Switcher is responsible for processing the different event codes it receives and switching to and from events accordingly. This also includes the information from the gameInfo struct which tracks the overall progress of the story.

4.4.2 Implementation

The implementation of the Event Switcher relies heavily on using switch statements. The following lines of code is a snippet from the eventSwitcher.c which implements this.



```
gameInfo eventSwitcher(gameInfo _branchInfo) {
   switch (_branchInfo.nextEvent) {
       case 0:
           _branchInfo = mainMenu(_branchInfo);
           break;
       case 100:
           _branchInfo = dayOne(_branchInfo);
           break;
       case 101:
           _branchInfo = dayOneCont(_branchInfo);
           break;
       case 102:
           _branchInfo = dayOneContTwo(_branchInfo);
           break;
    }
    return _branchInfo;
}
```

Figure 4.3. Partial Code of the eventSwitcher

`eventSwitcher()` is called from the `main()` function that is continuously looping until the `_branchInfo.end` is 1. If it is not 1, then the `eventSwitcher()` function switches between different events, saving the info from the events in between.

5 Gameplay

5.1 General Description

Doki Doki CMSC Club is a dating-sim inspired game that contains a turn-based battle system. It incorporates the intricacies of managing relationships between people and strategy during the battles.

5.2 Overall Flow of the Game

5.2.1 Description

The game flows through events depending on your actions, as each action has its own effect on the story. The game has different branches depending on your options, however it all converges to one point in the story.

5.2.2 Implementation

The main controller of the game is the struct named `gameInfo`, defined in gameInfo.h. It contains the main info that the game runs on, the struct includes the `end`, `nextEvent`, `errorCode`, `interestPoints` and `hearts` identifiers. `end` is a flag that signals when the game should end. `nextEvent` is an identifier that stores the event code of the next event. `interestPoints` is an array that contains the interestPoints of each character and hearts is `interestPoints`/100.

The game will run on a loop until end becomes 1, where the game will end. The loop is continuously calling the eventSwitcher() function which was presented in Section 4.4. After an event is called, it will then display the lines for that event and the options. The implementation of this was presented in Sections 3 and 4. Users can control the flow of the story through the arrow keys. The implementation for this will be explained in Section 6

Splash screens are presented throughout the game. They will run when the game starts, when the story has been started by the user, when a battle starts and when the game ends. The implementation of this can be found in hud.c, lines 31 - 142. Lines 31 - 47 contains the code for printing the splash screens which is as follows:

```
void fullScreenCentered(const char **line, int lineSize) {
   clear();

   //Get Terminal Size
   int row, col;
   getmaxyx(stdscr, row, col);

   //Prints line
   attron(COLOR_PAIR(2));
   for (int i = 0; i < lineSize; i++) {
       mvprintw(((row / 2) - floor(lineSize / 2)) + i, (col - strlen(line[i])) /
2, "%s", line[i]);
   }
   attroff(COLOR_PAIR(2));

   refresh();
   sleep(4);
}
```

Figure 5.1. Code for fullScreenCentered

da

`fullScreenCentered()` takes the arguments line and line size. `line` contains the array of strings and `lineSize` is the number of strings in the array. The size of the terminal was obtained and it is used in calculating where the line should start printing. `mvprintw()` moves the cursor to where the line should start printing which is calculated by the following statements: `row / 2 - floor(lineSize /2) + i` calculates the row where the splashscreen should start printing so that the entire thing will be centered, `col - strlen(line[i])` calculates the column where the line should start printing so that it will be centered. `attron(COLOR_PAIR(2))` and `attroff(COLOR_PAIR(2))` sets the color of these splash screens.

5.3 Battle System

5.2.1 Description

One of the main gameplay mechanics is the battle system. The battle system is driven by the calculation of random identifiers and the choices of the player. The concept of a turn-based system and the user interface was heavily inspired by games from the Pokemon series. The player would be given the choice of choosing between three general actions: **Attack**, **Evade**, and **Taunt**. Each option is renamed and redone depending on the opponent the player is facing, but would ultimately still do the same action. On every turn, there are three random identifiers that are to be calculated: the opponent's action, the opponent's success rate, and the player's success rate. The result of these three identifiers plus the desired action of the player would determine what would happen in the battle.

5.2.2 Implementation

The implementation of the battle system can be found at `battleSystem.c`. The main identifiers that drive the battle system are the following: `boss.move`, `bossSuccess`, `playerSuccess`, and `optionWindow.choice`. The information that the battle system will continuously manipulate is stored in `playerStruct` and `bossStruct`. Each opponent is equipped with different information and dialogue cues to make sure that each battle the player encounters would feel unique and dynamic.

The `boss.move` identifier, taken from `bossStruct`, is used to determine what action the opponent will take during their turn. The opponent will only have three possible actions, the same as the player.

The `bossSuccess` and `playerSuccess` identifiers are used to determined whether the opponent or the players action would be considered successful or not. The identifiers' values could range from 0 - 99. Each opponent has a specified skill identifier known as `boss.skill`. This identifier paired with the `bossSuccess` identifier, determines the outcome of the opponent's action.

Lastly, the `optionWindow.choice` identifier contains the decision that the player makes in regards to his/her desired action. The battle itself will continuously loop until the health of either the player or the opponent becomes less than or equal to 0.

```c
int bossBattle(int bossSelection, gameInfo _battleInfo) {
   //Setting Battle Info
   unsigned int action = 0;
   int battleOutcome = 0, bossSuccess = 0, playerSuccess = 0;
   bossStruct boss;
   playerStruct player;
   srand((unsigned int)time(NULL));
   int success = 0;

   //Setting Player Info
   player.health = 100;
   player.maxHealth = 100;

   //Selecting Boss Attributes
   switch(bossSelection) {
       case 1:
           player.damage = 20;
           boss.damage = 10;
           boss.health = 100;
           boss.maxHealth = 100;
           boss.skill = 50;
           boss.name = "Boss 1"
           break;
   }

   // Main Battle Loop;
   while (boss.health > 0 && player.health > 0) {

       // Generates random boss move
       boss.move = rand() % 3;

       // Calculates success rate
       bossSuccess = rand() % 100;
       playerSuccess = rand() % 100;

       // If boss attacks
       if (boss.move == 0) {
           // If attack succeeds
           if (bossSuccess >= boss.skill) {
               // If player chose to attack
               if (optionWindow.choice == 0) {
               }
           }
           // If attack misses
           else {
           }
       }

   if (player.health <= 0) {
       success = 1;
   }
   return success;
}
```

Figure 5.1 Sample code that explains the logic of BattleSystem.c

6 Game Controls

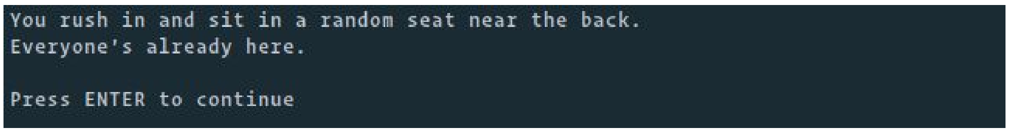
6.1 General Description

The main control scheme used during the game is through letting the users select options by using the *Arrow Keys*. This is done to improve user experience and interface. Since there are two key screens, different control layouts were created to better fit the user interface per screen.

6.2 Dialogue Screen

6.2.1 Description

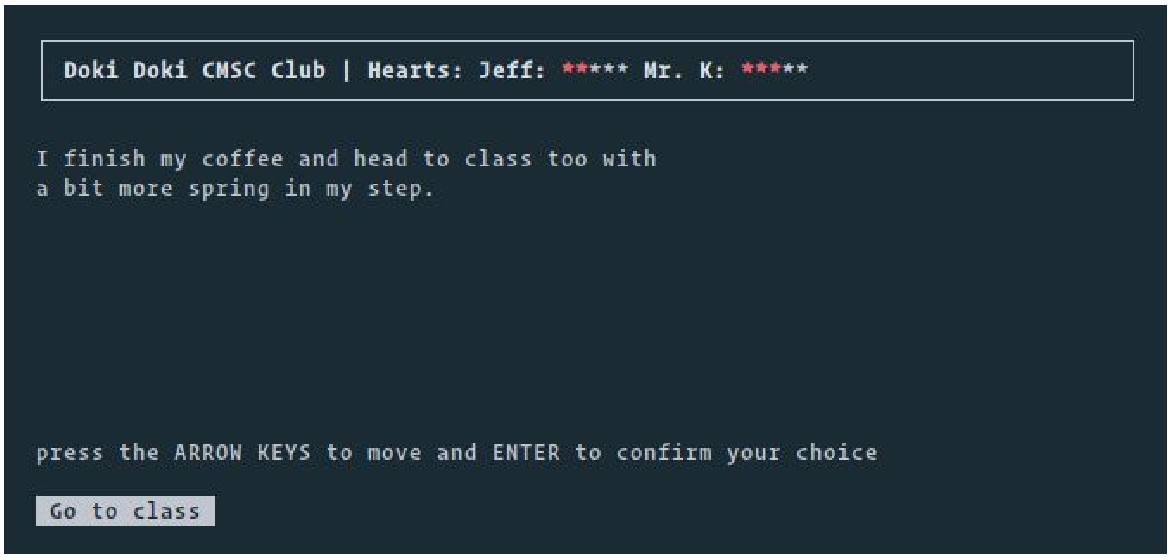
There are two methods to control the game when inside the dialogue screen. The first method is through letting the user press *Enter* or any key to proceed to the next lines of text. The following image shows how this control scheme appears in the game.



```
You rush in and sit in a random seat near the back.
Everyone's already here.

Press ENTER to continue
```

Figure 6.1. Prompting the user to Press ENTER to continue

The second method is to let the user select from a given set of options through the usage of the *Up* and *Down Arrow Keys*. If they are at the first option and the press the *Up Arrow Key,* the selected option will wrap around to the last option. The same is also true if they are at the last option.



```
Doki Doki CMSC Club | Hearts: Jeff: ***** Mr. K: *****

I finish my coffee and head to class too with
a bit more spring in my step.

press the ARROW KEYS to move and ENTER to confirm your choice

Go to class
```

Figure 6.2. Option control during dialogue screen

6.2.2 Implementation

6.2.2.1 Press ENTER to continue

To implement pausing the printing of lines and prompting the user to Press ENTER to continue, the following things were considered: When to pause the printing of lines and what to do after pressing enter. The following code, found at hud.c line 146 - 170 does this:

```
    int contentRow = 1, counter = 0;
    int contentRowHolder = 0;
    while (counter < lines) {
        if (!(strcmp(line[counter], ""))) {
            mvwprintw(contentWindow, contentRow + 1, 0, "Press ENTER to 
continue");
            wrefresh(contentWindow);
            wgetch(contentWindow);
            wmove(contentWindow, contentRow + 1, 0);
            wclrtoeol(contentWindow);
            contentRow = 1;
        } else {
            if (contentRow == 1) {
                wclear(contentWindow);
            }
            mvwprintw(contentWindow, contentRow, 0, "%s", line[counter]);

            if (strlen(line[counter]) > col) {
                contentRowHolder = (strlen(line[counter]) / col) + 1;
            } else {
                contentRowHolder = 1;
            }
            contentRow += contentRowHolder;
        }
        counter++;
    }

```
Figure 6.4. Code found at hud.c line 146 - 170

`counter` is used to track the amount of lines that has already been printed. `contentRow` specifies which row the line is to be printed and `contentRowHolder` counts the number of rows the current line has printed if the line has wrapped to the next row. `line` is an identifier that is passed to the function that contains an array of strings while `lines` is an identifier that was passed to this function that indicates the number of lines that are in the array.

To implement this, the loop checks if the current string does not have characters inside it, if it does not then that means that it is time to pause the printing of the lines and to prompt the user to press enter to continue. This is done using the `wgetch()` function in ncurses. If the line contains characters, then the line will be printed normally.

The main constructs used in implementing this are `if` statements and `while` loops.

6.2.2.2 Options Menu

The main point of emphasis on the options menu is the usage of the arrow keys to select the option. In addition to what was mentioned before, this method of selecting options removes the necessity to check for invalid inputs. The following code, found at hud.c line 191 to 222 implements this.

```
while (1) {
    for (int i = 0; i < options; i++) {
        if (i == returnValues.choice) {
            wattron(returnValues.optionWindow, A_REVERSE);
        }
        mvwprintw(returnValues.optionWindow, i + 3, 0, " %s ",option[i]);
        wattroff(returnValues.optionWindow, A_REVERSE);
    }
    choice = wgetch(returnValues.optionWindow);
    switch (choice) {
        case KEY_UP:
            if (returnValues.choice == 0) {
                returnValues.choice = options - 1;
            } else {
                returnValues.choice--;
            }
            break;
        case KEY_DOWN:
            if (returnValues.choice == options - 1) {
                returnValues.choice = 0;
            } else {
                returnValues.choice++;
            }
            break;
        default:
            break;
    }
    if (choice == 10) {
        break;
    }
}
```

Figure 6.5. Code found at hud.c line 191 - 222

`returnValues` is a struct created to hold the ncurses window and the choice of the user. `returnValues.choice` contains the selected choice of the user after they press the enter key. `option` is a identifier that is an array of strings and `options` is the identifier that contains the number of options that is in the array. Both of these are passed to the function from the event function.

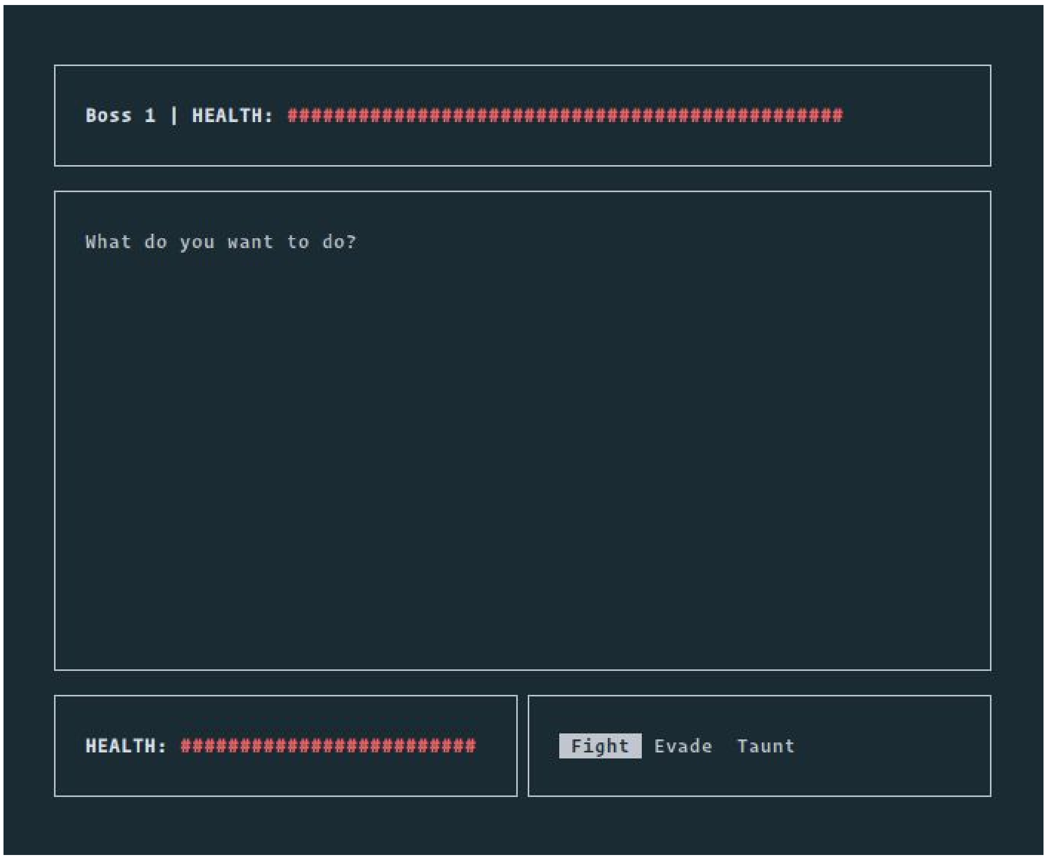
To implement this, the options are continuously printed until the user has selected an option by pressing Enter. Looping is done to show which option is selected and to highlight the selected choice correspondingly. To show the selected option that the user has selected, `returnValues.choice` is being incremented if arrow down is pressed and decremented if arrow up is pressed, unless if `returnValues.choice` is at the top or the bottom of the options list. If they are at the top or at the bottom, then they wrap around to the bottom or the top, respectively. If`returnValues.choice` is equal to the current string being printed, then implementing the player action menu for the battle screen, the code used was the same but modified to use the left and right arrow keys. en `wattron()` is used. `wattron()` applies the `A_REVERSE` attribute which reverses the colors of the background and the text, akin to highlighting it. When the Enter key is pressed, the value of `returnValues.choice` is returned to the event function that created the game screen.

The main constructs used here are `switch` and `if` statements, and `for` and `while` loops.

6.3 Battle Screen

6.3.1 Description

The main method of control when in a battle is through selecting actions through the use of the *Arrow Keys*. The following actions are the main options when in a battle - Fight, Evade and Taunt - however, these options can be changed depending on the event that precedes it.

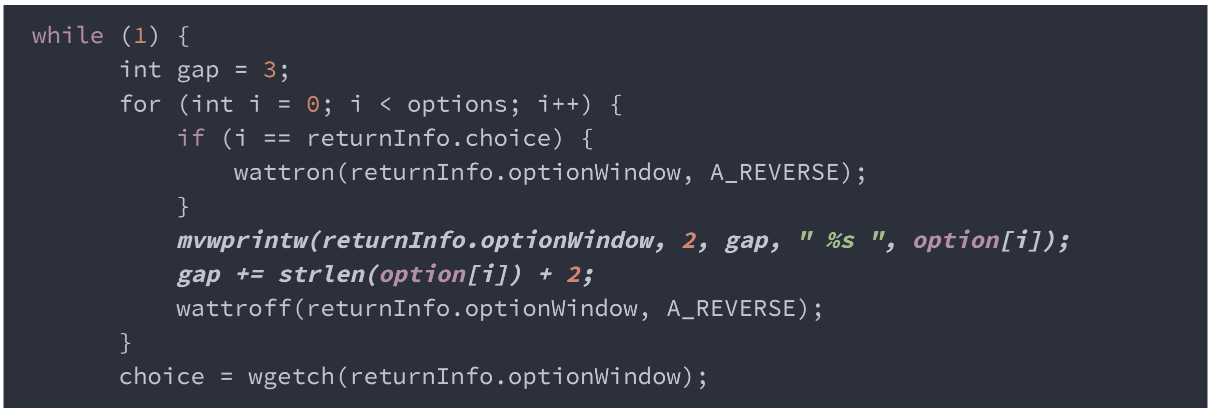


```
Boss 1 | HEALTH: ###############################################

What do you want to do?

HEALTH: #########################       Fight  Evade  Taunt
```

Figure 6.3. Prompting the user to do one of the three actions

6.3.2 Implementation

The implementation of the controls in the battle screen is exactly the same as the implementation of the controls in the dialogue screen when selecting options. To view how this was implemented, you may look at section 6.2.2.2 of this paper. However, this version of the menu prints the options on the same line, this adds a level of difficulty as the spacing needs to be taken care of as the row height cannot be simply added if a new line is present as they are on the same line. The following code takes care of this:



```
while (1) {
      int gap = 3;
      for (int i = 0; i < options; i++) {
          if (i == returnInfo.choice) {
              wattron(returnInfo.optionWindow, A_REVERSE);
          }
          mvwprintw(returnInfo.optionWindow, 2, gap, " %s ", option[i]);
          gap += strlen(option[i]) + 2;
          wattroff(returnInfo.optionWindow, A_REVERSE);
      }
      choice = wgetch(returnInfo.optionWindow);
```

Figure 6.4. Lines of code that takes care of the positions of the options.u

The italicized portion of figure 6.4 takes the length of the current string and adds it to the identifier named gap. 2 is also added to take care of the space between the string. This ensures that the options, with variable length, being printed will not overlap with each other and spacing will be maintained.