CS141 – 02

Prof. Edwin Rodríguez

06/01/2012

# Team Kelly

# “My CompSci Girl”

Team Members:

John Tran

Kwang Jae Jun

Kelly Nguyen

Mesh Chuong

Dominick Do

# Introduction

In this game, you are a Computer Science geek currently studying at Cal Poly Pomona. You don’t always shower and you’re the type of guy who would rather code than hang out with friends. Somewhere along your journey, you meet the girl of your dreams; and her name is Kelly. She studies Computer Science just like you, likes the same RPGs you do, and can even beat you in Dance Dance Revolution. The two of you have been dating for some time now and you think she is the one. You plan to propose to her. You got the ring and the girl, what could possibly go wrong?

One stormy day, after a long day of class, you get a text message from Kelly. It reads:

------------------------------------------------------------------------

------------------------------------------------------------------------

> I have your girlfriend locked up and hidden…she will be my coding slave for all of eternity!!! You will never get to her. I have hired her ex-boyfriends and they are patrolling the halls. Go home, for you will be forever alone!
>
> - Edwin Rodriguez

------------------------------------------------------------------------

------------------------------------------------------------------------

It’s getting dark and you can hear the thunder pounding throughout the night sky. There’s a storm brewing. It begins to pour rain. In a panic, you run into Building 8 to take shelter as your mind begins to run through a million questions…where could Kelly possibly be? How were you going to find her? The lights flicker. At the end of the hallway you see one of Kelly’s ex-boyfriends walking quickly towards you, staring you down. The storm outside shuts off the lights and you take the moment to run and hide. Edwin has Kelly’s ex-boyfriends guarding the building to keep you away from Kelly. She’s so close, and all you have to do is find her.

As all the pieces begin to come together, you realize what you must do. Reaching for your phone, you reply back to the previous text:

> --------------------------------------------------------------------
>
> --------------------------------------------------------------------
>
> I don't know who you are. I don't know what you
> want. If you are looking for ransom, I can tell you
> I don't have money. But what I do have are a very
> particular set of skills; skills I have acquired over
> a very long career of gaming. Skills that make me
> a nightmare for people like you. If you let my
> girlfriend go now, that'll be the end of it. I will not
> look for you, I will not pursue you. But if you don't,
> I will look for you, I will find you, and I will kill you.
>
> - Your worst nightmare
>
> --------------------------------------------------------------------
>
> --------------------------------------------------------------------

You reach into your backpack and find one copy of the Diablo III Collector’s edition and a mini keychain flashlight. Your mission is to free Kelly from the grasps of the evil Edwin. In order to complete your mission, you must avoid getting caught and beaten up by her ex-boyfriends. May the Java gods guide you on this mission to marry the girl of your dreams.

# Project Description

This is a team project in which a text-based game will be created using Object Oriented techniques. It will take place inside a building which is represented as a grid of eighty-one squares. In addition, there are nine equally spaced special squares that represent rooms. There are entities that represent a player, enemies, and power-ups which can be placed anywhere on the grid with the exception of the nine rooms. The player must move the player character around the building, checking the rooms for a briefcase. The building is pitch-black therefore the player can only see one square ahead of his position. The player must avoid being captured by the enemies patrolling the building.

At the start of the game, the player character is placed at the bottom left-corner of the grid. The enemies are randomly placed across the grid at least three squares away from the player character’s starting position. Also, the briefcase is randomly placed in one of the nine rooms. The player can only enter a room from the north-side. The player character and the enemies take turns accordingly, with the player having the first move. At every player’s turn, the character can look, move, or shoot. The player character has night-vision goggles that allow him to see two squares from his position. A player may ‘look’ in any direction which will allow him to see a total of two squares ahead of his position. The ‘looking’ ability will display either a ‘clear’ or ‘enemy ahead’ signal. It can also only be done once per turn. A player can move one square in any available direction. A player can also shoot in any direction if he has ammo.

Once it is the enemies turn, they will check to see if the player character is in any adjacent square. If the player is in an adjacent square, the enemy will capture the player. Otherwise, the enemy will move one square in a random direction.

In the game, there are three power-ups that are randomly placed across the grid. There is an additional bullet that will grant the player an additional bullet only if the player has already used his initial bullet. Another power-up is the radar, which will automatically display the location of the briefcase on the screen.  The last power-up is the invincibility power-up which will grant the player invulnerability to capture from the enemy for five turns.

The player has three ‘lives’.  The player loses a life every time the enemy captures him.  The player will be repositioned back to the initial position after losing one life.  If the player loses all three lives, then he or she loses and the game is over.  The player can save and quit the game at any time and come back to reload a previous saved game.

# Design Approach

Our goal was to develop a hierarchy of classes. We have a grid map which would display our other classes and a Game Engine to run the functions of the game. We have classes such as Cell which is subclassed by Room, Unit and Item. The Grid has a two-dimensional array of the Cell, a list of Rooms, Items, and Enemy, and an object to the Player. It will take input from GameEngine to decide their positioning. Room position is always final, and nothing can take the place of the Cell Room, the room can only be entered from the North by the Player. The Items will be randomly placed on the grid, three items are available - Bullet, Invincibility, Radar. The Enemy can occupy the same cell as Item, but it will not erase the Item. When the Enemy leaves the cell the Item will appear again. The Units are composed of the subclasses Player and Enemy. The Player is positioned on the bottom left corner, the player can move, shoot, and look. Once the player moves or shoots their turn is over, when the player’s turn is over the Enemies turn will take place. The Enemy is placed randomly in Grid but not close to the Player. When it is the Enemy’s turn they will first search the surrounding cells for the player, if the player is in the surrounding cell, then the player will die. For artificial intelligence, if the enemy locates the player and is right next to the player, the Enemy’s next turn will follow the Player. If the Enemy does not find the player it will move in a random direction as long as that direction is in the bounds of the grid. The game will end when the player enters the room that contains the goal.

# Discussion of Implementation

We first implemented the class Cell, which is an abstract class. This Class represents each cell of the Grid. It has three fields; symbol, empty, and visible. Symbol contains the current symbol of the cell, if the cell is empty it will contain a space. For empty, this indicates if the cell is empty, if nothing is on the cell this field is false, otherwise it is true. For the boolean visible, this field represents if the target cell is visible. These three fields have a setter and getter so that we can change and check the status of the target cell. The cell has a behavior of getSymbol(), it does not simply return the symbol of the cell, it returns whether or not the cell is visible. There are three classes Room, Item, and Unit extending the Cell. Room has a boolean field that indicates if the room contains Kelly, who is the person being rescued. Its constructor initializes its super class by calling the special method super(char) that passes the Room symbol ‘R’. The next subclass of Cell is Item and this subclass is an abstract class as well. Item contains fields that dictate the positioning of an Item, such as rowPosition and colPosition. It has a boolean field obtained, which represents whether or not the Item has been picked up by the user. Its constructor uses the special method super(char) to pass the Item symbol to its superclass. This class has an abstract method consumeThisItem(Player); each item will have its own version of this method. There are three subclasses that extends Item, which are Bullet, Radar, Invincibility. These classes implements the abstract class such that the Bullet will add the Player’s ammo count by one, the Radar will show the room with Kelly, and the Invincibility will the Player invincible for the next five turn.

The Class Unit is also an abstract class. It has fields that dictate its positioning on the grid, and has a boolean field of alive to represent the life and death of the unit. The construct for this class uses super(char) to pass the Unit symbol to its superclass. The Unit has an abstract method of move(Cell[][], char), which will be used by its subclasses since each of the classes has different behavior to move. The next abstract method boolean isWrongDirection(Cell, char), will determine whether or not the move of the Unit was valid in the different conditions. The method move(char dir) moves the unit by by taking a direction and then moving the position on the grid accordingly. The method moveBack(char dir) will move the Unit back to its previous position, this method is called upon when the Unit makes an invalid move such as trying to move onto another Unit or moving out of the Grid. The abstract class Unit has two subclasses; Player and Enemy. The method move(Cell[][], char) in Player will check to see if the the target cell is empty and if so, it will move there. If the cell is not a valid move then the method will return wrong direction. If the Player enters the Room from the north, the method will check if the room contains the flag. And if the cell is occupied with an Item, then the Player will obtain the item. The method boolean isWrongDirection(Cell , char) will first check to see if the target cell is empty, then it will check to see of the cell is occupied by an Enemy or a Room, if the player enters the wrong side of the room or of the flag is not inside of the room the boolean will return true. This value will determine whether or not the Player will get to redo its move turn. The class Enemy  contains a boolean field foundPlayer, which will be used to determine whether or not the Enemy has discovered the Player. The constructor for enemy simply uses the special method super(char) to pass the enemy symbol to its super class. For Enemy’s method move(Cell[][], char) it move the Enemy in the given direction (which is randomly generated). It then checks if the move was a valid move by using the method isWrongDirection(Cell, char). This method checks to see if the cell is occupied by an Enemy, Room, Flag, or Player, if so the method will return true. The method lookForPlayer(Unit) will search the surrounding cells for the player. The method catchPlayer(Unit) will catch the player and cause them to lose a life. When the Player no longer has any more lives then method killPlayer(Unit) is used to kill the the player. This class also has getters and setters for important fields such as setting whether or not the Enemy has found Player.

The class Grid creates the grid map and locates rooms, items, enemies, and player on the map with the method createGridMap(). For each turn of the player, this class will move the player or perform the actions shoot and look, and the method moveOfPlayer(char, char) takes care of this functionalities. For each of the enemies turn, the method moveOfEnemy() will be invoked to check if the player is around and to move in a random direction. When a player or an enemy moves, its current cell will be assigned with a new Cell and the target cell to move will be assigned with the player or the enemy. After these two kinds of the turn are done, the visibility of the grid map will be reset so that the user can see the updated screen and the method setGridMapVisibility() takes care of this task. First of all, this method will set the visibility of the all cell equal to false, then make the rooms visible using the list of the room, and finally, gets the vision of the player. In this class, there is a method called refreshItemCell() and this method prevent the item symbols disappeared from the grid map when an enemy move on the the item and move away. For AI, the Grid marks those enemies right next to the player with the method markAiEnemy() and it gives the direction the player moved toward for the next turn of those marked enemies.

The class GameEngine is to interact the user and the game. It takes all of the inputs and distributes each behaviors depending on the inputs. Also, it keeps track the game status such as if the player’s turn is over and the game is over. One of it main functionalities is storing the direction the player moved toward for AI feature.

The class UserInterface creates the output to the user. This is what the user sees and how they can interact with the program. The UserInterface starts out by initializing a field GameEngine. Its default constructor is empty. The most important method used in this class is the selectMainMenu(Scanner), this class is a series of switches which will implement whatever choice the user gives - whether to start a new game, to load a new game, how to play, debug mode or quitting the program. These switches interact with the GameEngine to run the selected choices.

The Main class creates a UserInterface and is basically the key to start the program. When the main class runs, it creates a domino effect of initializing the other classes.

# Testing Approach

Our method for testing each use case was to first compile the program in Eclipse Indigo version SR2 on 3 different machines with Java 1.7 installed for consistency. From there, we entered debug mode in the game and ran each scenario from our list of use cases. When a test would fail, we would mark it down and our entire group would go over the code and try to come up with ideas to try in order to resolve the problem. Once some possible solutions are suggested, we would implement them and test again to see if the individual test would pass. If it didn’t, we moved on to try the next suggestion. When a test would pass on all the test environments, we would mark it as passed. Many times a test case would fail from a simple coding flaw which would be easily fixed. Once a problem became marked as resolved, our testers would once again compile the new code on multiple machines and retest all of the use cases from start to finish in order to ensure that a new bug wasn’t introduced while fixing the previous one. By eliminating bugs one by one in this systematic manner, we were able to reach the current state of the program which passes all the following test cases.

# Test Cases

| Use Case | Actors | Goal | Pass |
|---|---|---|---|
| **Player location at the start of the game** | Player | The user prompts the system to either “Start Game” or “Debug Mode”. This will begin the game and place the Player in the leftmost bottom corner of the grid. Player is placed in (Row 8, Column 0) at start of the game. | ✓ |
| **Enemy locations at the start of the game** | Enemy | The user prompts the system to either “Start Game” or “Debug Mode”. This will begin the game and randomly place all of the instances of Enemy in randomly allocated locations on the map. Each Enemy will occupy its own cell at the start of the game. | ✓ |
| **Item locations at the start of the game** | Invincibility Item, Radar Item, Bullet Item | The user prompts the system to either “Start Game” or “Debug Mode”. This will begin the game and randomly place the Invincibility Item, Radar Item, and Bullet Item onto randomly allocated locations on the map. | ✓ |
| **Briefcase (Kelly) location at the start of the game** | Briefcase/Kelly | The user prompts the system to either “Start Game” or “Debug Mode”. This will begin the game and randomly allocate the Briefcase in one of the nine Room locations on the Grid. | ✓ |

| | | | |
|---|---|---|---|
| **Player moves north from row (1-8)** | Player | Typical/Basic movement of the Player. This use case is for the Player to move north when the Player is within Rows 1-8. Row 0 is excluded because that is a special case--Player cannot move north if it is already at the very top of the map. | ✓ |
| **Player moves south from row (0-7)** | Player | Typical/Basic movement of the Player. This use case is for the Player to move south when the Player is within Rows 0-7. Row 8 is excluded because that is a special case--Player cannot move south if it is already at the very bottom of the map. | ✓ |
| **Player moves east from column (0-7)** | Player | Typical/Basic movement of the Player. This use case is for the Player to move east when the Player is within Columns 0-7. Column 8 is excluded because that is a special case--Player cannot move east if it is already at the very right of the map. | ✓ |
| **Player moves west from column (1-8)** | Player | Typical/Basic movement of the Player. This use case is for the Player to move west when the Player is within Columns 1-8. Column 0 is excluded because that is a special case--Player cannot move west if it is already at the very left of the map. | ✓ |

| | | | |
|---|---|---|---|
| **Trying to go north at row 0** | Player | The player is placed in (Row 0, Column 0-8). The user attempts to move the Player north even though there is no cell space available. The user should be prompted by the program that they have selected an invalid move input. The user will be asked to press any key. The grid should be printed again with the Player in the same position--as well as all of the other enemies--as previous to when the player attempted to make a wrong direction. | ✓ |
| **Trying to go south at row 8** | Player | The player is placed in (Row 8, Column 0-8). The user attempts to move the Player south even though there is no cell space available. The user should be prompted by the program that they have selected an invalid move input. The user will be asked to press any key. The grid should be printed again with the Player in the same position--as well as all of the other enemies--as previous to when the player attempted to make a wrong direction. | ✓ |

| | | | |
|---|---|---|---|
| **Trying to go east at column 8** | Player | The player is placed in (Row 0-8, Column 8). The user attempts to move the Player east even though there is no cell space available. The user should be prompted by the program that they have selected an invalid move input. The user will be asked to press any key. The grid should be printed again with the Player in the same position--as well as all of the other enemies--as previous to when the player attempted to make a wrong direction. | ✓ |
| **Trying to go west at column 0** | Player | The player is placed in (Row 0-8, Column 0). The user attempts to move the Player west even though there is no cell space available. The user should be prompted by the program that they have selected an invalid move input. The user will be asked to press any key. The grid should be printed again with the Player in the same position--as well as all of the other enemies--as previous to when the player attempted to make a wrong direction. | ✓ |
| **Entering a room at from its north side** | Player, Room | Player is located above one of the nine rooms. The user inputs a South move direction. The user is prompted by the system that the room is either empty or not empty. If it is empty, the game continues. If it is occupied with the correct item then the game is over and the user has won. | ✓ |

| | | | |
|---|---|---|---|
| **Entering a room at from its south side** | Player, Room | Player is located below one of the nine rooms. The user inputs a North move direction. The system should prompt the user that the direction is invalid. They will be asked to press any key. The Player and all of the enemies remain in the same position as they were previous to when the user inputted an invalid direction. | ✓ |
| **Entering a room at from its east side** | Player, Room | Player is located to the right of one of the nine rooms. The user inputs a west move direction. The system should prompt the user that the direction is invalid. They will be asked to press any key. The Player and all of the enemies remain in the same position as they were previous to when the user inputted an invalid direction. | ✓ |
| **Entering a room at from its west side** | Player, Room | Player is located to the right of one of the nine rooms. The user inputs a East move direction. The system should prompt the user that the direction is invalid. They will be asked to press any key. The Player and all of the enemies remain in the same position as they were previous to when the user inputted an invalid direction. | ✓ |
| **A Player looks north** | Player | By default, the Player can always see one cell to the north, south, east, and west of them. If the user inputs a North Look Direction, then they should be able to see two spaces north of them while all of the other directions--south, east, west--only one space of visibility. | ✓ |

| | | | |
|---|---|---|---|
| **A Player looks south** | Player | By default, the Player can always see one cell to the north, south, east, and west of them. If the user inputs a South Look Direction, then they should be able to see two spaces north of them while all of the other directions--north, east, west--only one space of visibility. | ✓ |
| **A Player looks east** | Player | By default, the Player can always see one cell to the north, south, east, and west of them. If the user inputs a East Look Direction, then they should be able to see two spaces north of them while all of the other directions--north, south, west--only one space of visibility. | ✓ |
| **A Player looks west** | Player | By default, the Player can always see one cell to the north, south, east, and west of them. If the user inputs a West Look Direction, then they should be able to see two spaces north of them while all of the other directions-- north, south, east--only one space of visibility. | ✓ |
| **A Player inhabits the cell North of an Enemy** | Player, Enemy | The Player moves to the cell North of an Enemy. The user will be prompted that the Player has been caught. The Player’s number of lives will be decremented by 1. If the Player has 1 life in this situation, then the game will end. | ✓ |

| | | | |
|---|---|---|---|
| **A Player inhabits the cell South of an Enemy** | Player, Enemy | The Player moves to the cell South of an Enemy. The user will be prompted that the Player has been caught. The Player’s number of lives will be decremented by 1. If the Player has 1 life in this situation, then the game will end. | ✓ |
| **A Player inhabits the cell East of an Enemy** | Player, Enemy | The Player moves to the cell East of an Enemy. The user will be prompted that the Player has been caught. The Player’s number of lives will be decremented by 1. If the Player has 1 life in this situation, then the game will end. | ✓ |
| **A Player inhabits the cell West of an Enemy** | Player, Enemy | The Player moves to the cell West of an Enemy. The user will be prompted that the Player has been caught. The Player’s number of lives will be decremented by 1. If the Player has 1 life in this situation, then the game will end. | ✓ |
| **A Player shoots an Enemy from the Enemy’s north side** | Player, Enemy | The Player is above and in the same column as the Enemy. The user chooses the Shoot input and South direction. All enemies in that entire column are killed and the ammunition count of the Player is decremented by 1. This test is used to ensure that no matter from what direction the Enemy is getting shot from, they will be killed and removed from the map. | ✓ |

| | | | |
|---|---|---|---|
| **A Player shoots an Enemy from the Enemy’s south side** | Player, Enemy | The Player is below and in the same column as the Enemy. The user chooses the Shoot input and North direction. All enemies in that entire column are killed and the ammunition count of the Player is decremented by 1. This test is used to ensure that no matter from what direction the Enemy is getting shot from, they will be killed and removed from the map. | ✓ |
| **A Player shoots an Enemy from the Enemy’s east side** | Player, Enemy | The Player is to the right of and in the same row as the Enemy. The user chooses the Shoot input and West direction. All enemies in that entire row are killed and the ammunition count of the Player is decremented by 1. This test is used to ensure that no matter from what direction the Enemy is getting shot from, they will be killed and removed from the map. | ✓ |
| **A Player shoots an Enemy from the Enemy’s west side** | Player, Enemy | The Player is to the left of and in the same row as the Enemy. The user chooses the Shoot input and East direction. All enemies in that entire row are killed and the ammunition count of the Player is decremented by 1. This test is used to ensure that no matter from what direction the Enemy is getting shot from, they will be killed and removed from the map. | ✓ |

| | | | |
|---|---|---|---|
| **A Player enters the correct room** | Player, Room | The user is prompted by the system that they have found the desired item. The system will prompt the user that they have completed the game. | ✓ |
| **A Player enters the incorrect room** | Player, Room | The Player enters an empty room.  The user is prompted by the system that the room is empty. The game continues as normal. | ✓ |
| **A Player picks up an Invincibility Item** | Player, Invincibility Item | This power-up will grant the geek invulnerability for five turns.  The Player occupies the same cell as an item (symbolized by an ‘a’). This renders the Player invincible and therefore at an advantage during this time in the game. | ✓ |
| **A Player picks up a Bullet Item (should increase ammo count)** | Player, Bullet Item | The Player occupies the same cell as an item (symbolized by a ‘d’). The ammo count that is displayed on the screen should increment by 1.. | ✓ |
| **A Player picks up a Radar Item** | Player, Radar Item | The Player occupies the same cell as an item (symbolized by a ‘p’). The next time the grid is printed out, the user can see the location of the “flag” (Kelly). The “R” symbol on the room with the desired item will change to a “K” to indicate this. It will stay like this throughout the remainder of the game. | ✓ |
| **Shooting with no ammo** | Player | When trying to shoot with no bullets remaining, the player should be presented with a status message indicating the player is out of bullets before losing a turn. | ✓ |

| | | | |
|---|---|---|---|
| **Kill invincible player** | Enemy | If a player still has more than 1 invincible move count remaining, they should not die to an enemy. | ✓ |
| **2 Enemies moving to same cell** | Enemy | When 2 enemies randomly move to the same cell, the one to move last will move back to its original position. | ✓ |
| **2 Items cannot spawn on same cell** | Item, Cell | Items spawn on a for loop which first check if a cell is empty.  If an item is already on the cell, it will run again until it finds an empty cell to spawn on. | ✓ |
| **Enemies cannot spawn in certain cells** | Enemy, Cell | Enemies must spawn a certain radius away from the player’s spawn and cannot spawn on top of a room.  During the random placing of enemies, there are checks to ensure the cell is empty as well as not in the rows and columns closest to the player | ✓ |
| **Selecting an invalid command option** | User | When the User Interface asks for a command (move, look, shoot), or direction (w,s,a,d), if anything other than the possible keys is pressed the user should be presented with an error message and be asked again for a valid input. | ✓ |
| **An enemy steps on an item** | Enemy, Item | The cell’s symbol will be replaced by the Enemy’s symbol, “E”.  Enemies and items are allowed to share the same cell, in our game the Enemy’s symbol will override the Item symbol on the cell. | ✓ |
| **An enemy steps off of an item** | Enemy, Item | When an enemy leaves a cell containing an item, the cell will restore the item’s symbol accordingly so that the Player knows an item is there. | ✓ |

| | | | |
|---|---|---|---|
| **A player steps on an item** | Player, Item | Once a player moves to a cell that contains an item, the item will be consumed and disappear from the grid. The Player will gain the corresponding effect depending on what the item symbol was. | ✓ |
| **Artificial Intelligence for Enemy** | Enemy | Enemies normally move in a random direction. With A.I., they will mimic the player’s moves as long as the Player was 1 row or column cell away. By breaking line of sight (turning a corner), the Enemy will revert back to random movements. | ✓ |
| **Player Caught** | Player | Players get caught if they are exactly 1 cell north, south, east, or west of an enemy. A Player starts the game with 3 lives, when they are caught they lose a life. If they only have 1 life remaining, they will die instead. | ✓ |
| **Player Death** | Player | A player dies when they get caught with only 1 life remaining. The game ends at this point with a message indicating so, as there are no more lives to continue playing with. | ✓ |
| **Enemy Death** | Enemy | An Enemy dies when a Player shoots them. This will call the Enemy’s die method and remove them from the grid as well as the array enemyList. | ✓ |

| | | | |
|---|---|---|---|
| **Can kill multiple enemies with a single bullet** | Bullet | Bullets should be able to kill an enemy and continue travelling to kill any other enemies in the same row or column. As long as there are no items or rooms in the way, all enemies in a bullet’s path should be killed. | ✓ |
| **Complete Game** | Player | Once the player has found the goal in the room, the game ends with a message to the user. A message will print out to indicate the game is completed. | ✓ |
| **A player and enemy attempt to move to the same cell** | Enemy | Since enemies move after a player moves, if a player enters a cell the enemy would have moved to, the enemy will kill that player because the first thing it does is check for a player before moving. | ✓ |
| **Cannot shoot through rooms or items** | Bullet | Bullets stop travelling once they reach an item, room, or go out of bounds | ✓ |
| **Rooms must spawn in certain cells** | Room | Rooms always spawn in the same cell positions. There are 9 rooms total, displaced evenly throughout the map. There is one room in the center of each 3x3 group of cells on the 9x9 grid. | ✓ |
| **Debug Mode** | Game Engine | An instance of the game will run which turns off any visibility options, revealing the entire map. Debug mode is used to make sure items, enemies, and rooms are all behaving and spawning in places where they should. This is easier done when everything is visible, so debug mode should disable the default visibility options. | ✓ |

| | | | |
|---|---|---|---|
| **User saves the game** | Game Engine | The user saves the game and the save file is able to be accessed.  The user prompts the system to save the game. The game is saved and listed into the save files list. This list can be accessed by the user later. | ✓ |
| **User loads the game** | Game Engine | The user prompts the system to load a game. The user chooses from a list of game saves. The game starts up at the point in the game where the user was previously playing at. | ✓ |
| **User quits the game** | Game Engine | The user prompts the system to quit the game. The program shuts down.. | ✓ |

# Discussion/Suggestions for Improvement

While working on our project, we found the limited time caused us to forego several features that we would have liked to implement in our final program. These are some of the extra elements that we decided we couldn’t complete in time but could significantly improve our program.

1. “Press any key” without having to press enter afterwards
2. More unique looking grid
3. GUI/Game art
4. Multiple save/load slots
5. More levels
6. Display high scores

To start off, an improvement would be to use the “press any key” option provided to us by the Professor. This option would have allowed the user to “press any key” instead of having to press “Enter” after every key. In its current state, the user has to press additional key inputs to do actions in the game, which could be seen as unnecessary and inefficient.

Another improvement would be to create a more unique grid layout instead of the simple print out we used. For the game, we used a simple grid as seen in the in class example that just prints out the characters and uses spaces and brackets to separate the grid squares. A more unique and or elaborate grid would be one that uses lines to outline the actual boxes of the grid.

Implementing a GUI would improve our game immensely through the use of creative visuals to match our game theme and title. We had originally planned on implementing GUI by completing the text-based program early. However, we found that GUI was a lot harder than we had anticipated. Because of this, we were unable to learn the material on our own to implement GUI.

Creating multiple save/load slots would allow the user to choose which saved game he or she wants to load and play. This would be an addition to the current one save/load slot that we have on our game.  A prospective idea would be to have the option of multiple save/load slots so that when all the slots are taken, the user may choose which slot he or she wants to be overwritten.

The option of creating more levels would make the game harder for the player.  As the number of level increases, the number of enemies will increase by one.  For example, level one would have six enemies, level 2 would have seven enemies, etc.  With this addition, the game would be more competitive and fun for the user.

Displaying the high score would be a possible addition of improvement.  By displaying the high score, it lets the user know how many moves were taken to beat the game in this level.  The user will then be able to plan accordingly in order to beat out the high score and set in place a new high score.  The high score will be the least number of moves taken to beat the game in that level with that particular setting.