Group Project Report

Team Deeh-Aeh-Squared+Kay

Members:

Allen Nagtalon

Angie Chong

Dulce Nava

Karen Chung

**Table of Contents**

**Section Title**

**Page**

Introduction/Motivation ……………………………………………………………………… 3

Problems ……………………………………………………………………………………… 5

Approach / Explanation of the Code …………………………………………………….... 8

GUI ………………………………………………………………………………………….... 12

Testing/Results ……………………………………………………………………………… 15

Conclusion …………………………………………………………………………………… 22

Intro /Motivation

Our team, Deeh-Aeh-Squared, were given the task to create a text based game as our class project. This game takes place inside a building and it is required that have a grid of 81 squares. Each of the squares are used to represent a position of the different objects such as the player’s avatar, enemies, power-ups, etc. The player character is a spy whose mission is to attain a briefcase that is located inside one of the 9 special squares that represents rooms. In the pitch black building, 6 ninjas should be roaming around guarding the rooms. These ninjas should be randomly located throughout the grid map while the player starts from the bottom left corner. The ninjas should be placed 3 squares away from the spy’s location while the briefcase is randomly located in one of the 9 rooms in the beginning of the game. The briefcase can only be grabbed from the north side of the grid. With the map being pitch black, the spy should not be able to see anything so if the player is equipped with night vision goggles, he is able to see two squares ahead of his position. The spy is the first one to move to a space and then the ninjas after. The spy is equipped with a gun but it is only allowed to have one bullet. At every turn, the spy is able to move one square in any space that is available in any directions and shoot if he hasn’t used the ammo yet. The spy is able to “look” at any direction and is allowed to do the following actions: move one square in any available direction and shoot in any direction if there is still ammo. If “looking” is used in the turn, it will either result with a “clear” or “ninja ahead” signal. This action of “looking” can only be done once per turn. In the start of every turn, the ninjas will check if the spy is located in an adjacent square. If he is, then it will result with the ninja killing the spy. If not, then the ninja will move one space in a random direction. In the game, there are also three power-ups on the board that are randomly placed. Additional bullet, invincibility, and radar will be spread throughout the board. The additional bullet will give the player more bullets but if the player still has his first bullet, it will have no effect. The invincibility power-up will give the spy the power to survive getting killed for five turns. The radar power-up will grant the player to automatically show the location of the briefcase by showing the room number of where the briefcase is located. The player has 3 lives and every time the player gets killed, it will restart again from his initial position. If the player loses all his lives, it is game over.

**Problems:**

Being a challenging project, we’ve been stumped more than once and ran into several problems that we were eventually able to fix. Being a group project, communication was a main problem we had very early on. We had 4 members in our team and as we talked about meeting up, we realized that our schedules classed severely as the only time available were Tuesdays or Thursday at a specific time. We knew that our team wouldn’t be able to work on the code together thus we decided to split up the parts and work on it separately to be more time efficient. Unlike most groups, we did not use GitHub, which could have helped us be more organized but we did not know of it until later on. Instead, we opted to communicate using Google docs, Google drive, texting and emailing to discuss, upload, and share our work.

Deciding to split up the work, it was difficult to properly understand the code each of us had done since we all had a different and personal style. Most of the time, we weren’t able to fully understand the code done by the other members so when we met up, we made it so that the person who coded it would go over step by step and properly explain to all of us. Usually that would be enough to clear things up and we’d be able to reach a common ground. However, difficulties arose again when it was time to add all the individual codes together into the main program. We would always have to tweak and modify some sections and several times it would not work until a couple of tries.

An additional problem we faced were misunderstandings regarding the work given to us. There were times where everyone on the team was assigned something to do but misheard or misunderstood the goal which resulted with a code that was completely unnecessary. It was a waste of time to create a code we didn’t need, so we made sure to fully understand what we wanted from each other when we are given a task to do.

Another bump we ran into was during the process of creating the GUI. In the beginning of the process, we used a different computer monitor to make the basic graphics of the GUI such as the grid board and pixels. Since we used a monitor, we sized everything accordingly to it instead of our laptop screen resolution. Because of our mistake, once we finished creating the GUI, the game would not fit on our laptop screen. The text box would be completely cut off below the laptop’s taskbar so there was no way to properly see it.

Moving to the technical difficulties such as problems we had with the program and code itself, one was that if any of the enemies relocated in a way that blocked any valid movement from the player, he would then be stuck and would not be able to move during his turn. This problem was prevalent anywhere on the board so if the player was in the middle and 4 ninjas were each facing him, the player would not be able to move because he would be trapped. However, this problem occurred the most near the corners or right below the rooms. For the corners, it was the easiest location because only two ninjas were needed to trap the player from moving any further. With the situation near the rooms, if the player was directly below the room and ninjas were blocking him from the left and the right, he would be unable to move from his spot during his turn.

Furthermore, enemies being trapped also occurred. There would be cases when the ninjas would cluster together into a group that would result with them being unable to move during their designated turn. An example for this would be if one ninja’s position was at the top right corner and another was one right next to it on the left side. The ninja to the left would also have a room be right below him. With the two ninjas being next to each other at the corner, imagine another ninja one space to the left. The program will scan and make it move towards the right. Then it will go to the next ninja and command it to move. This is where the problem arose as the ninja in the middle would be sandwiched between the ninjas from his right and to his left as well as the room blocking his way from below. If this situation occurred, the game would crash and the player wouldn’t be able to input any new commands.

Another challenging problem we had was learning how to even create a GUI. No one in our team knew how to create a GUI or had any basic knowledge about it so we basically had to start from scratch. The only way we were able to get any sort of clue to create the GUI was by reading the chapters from the textbook. Even then, we had to do several outside research on our own to fully understand the concept. This was a challenging and time consuming experience because every time the GUI wouldn’t compute, we’d had to restart from the beginning to figure out what went wrong because we didn’t have a strong foundation on the subject.

**Explanation of the Code:**

The structure of our coding for this project mimics the same structure that we used for the Memory Game assignment from earlier in the quarter. Our main method, located in the Main class, creates a new interface that manages the external state of the game—either text-based or graphic-based, depending on the user’s choice—with a new GameEngine that controls the internal state of the game, such as the board, lives, etc. In each instance that the user must be prompted to input a value or something must be printed to the screen for the user to respond, the interface would execute the appropriate methods and receive the inputs. From there, the selected interface would call the appropriate GameEngine method to execute actions of the player’s choice.

The code for the GameEngine can be split into a few sections based on the actions that the player has access to in the game: looking, moving, shooting, saving, loading, and mode selecting. Each separate section interacts with the GameBoard class to move, spawn, erase, etc. objects on the board to reflect the actions that the user chooses. The focus for on methods follows this continuous flow to efficiently relay information between the user and the computer—think of it as a two-man conversation. When one person puts an input, the other responds accordingly based on the input, to which the initial speaker responds, and so on.

As for our take on how to deal with managing the board, we decided it would be best to group all map entities under one umbrella superclass called MapObjects. By making all the unit object types extend this MapObjects class, we could simply maintain the entire board in one two-dimensional array of size 9x9. This multi-dimensional array, saved as a field called “grid” in the GameBoard class, could be called throughout the class to make moving objects, erasing objects, altering object attributes, etc. easier. For example, in the case of moving, we could simply take the player’s position, set it to equal null, and set the player on the space relative the user’s input choice (up, down, left, or right).

For all stationary objects, we decided that positions would be trivial to keep track of. In the case of power-ups, it would be pointless to use the position of the power-up to compare with the player’s position, because, with the way we coded it, the player would essentially be deleting the power-up in the process of trying to acquire it. Instead, we decided it’d be much more efficient to activate power-ups/check rooms through logic by contradiction. To give an example of this, let’s assume the case of the player moving onto the room space that happens to contain the briefcase. In this case, when the player moves into the space, the player will have essentially deleted the briefcase from the map. To most, the initial thought is that this would be extremely problematic; however, this makes the job checking for a win much easier. If the player is occupying the briefcase room, then there should essentially be no briefcase on the map; therefore, after every move, our code scans through each space on the grid and looks for the briefcase. If the system can’t find the briefcase, it means the player already has it. In situations like these, don’t require a coordinate comparison to manage the interactions.

However, for moving entities, we felt that it was more essential to keep track of their location. Because of this, both the Player and Enemy classes have an int row and int column field to keep track of said location. Keeping track of the position for these objects—especially the Enemy objects—makes their actions and interactions much easier to code. One such example is the problem we had with enemies moving. In the game, there are multiple enemies on the board at a time, unlike the character; instead of moving just one object, we have to go through the process of moving every enemy. This becomes problematic trying to make enemies move relative to their initial location if we don’t *have* that initial location. Without keeping track of the position through fields, we would essentially have to scan the board six times for six instances to get the coordinates of each enemy—even more so in later levels, when the number of enemies is incremented by one for every level. As such, our team decided it’d be much easier to save the coordinates. This made functions such as saving and loading especially efficient.

In our game, saving and loading does not rely on one sole “master” save file; the user is free to make multiple save files of varying file names. When executing the save method, the user is prompted to input the name of the save, with which the Virtual Machine will create a new File object with. From there, we utilized the Serializable interface with a separate object class called Data. What this object type stores is numerous primitive data types and primitive data type arrays that store the internal state of the game; this includes such attributes as object locations, player lives, player bullets, etc. The Virtual Machine then takes this newly formed Data object and serializes it into the aforementioned file to be reconstructed at a later time, through the loading methods.

We did run into some inconsistency issues with the saving and loading methods, however. Between maintaining the same type of data types to maintaining the default directory of said save files, the text-based interface and GUI suffered major differences that made operating the game less overall user-friendly. While we were not able to fix said problems within the time frame of the project, we hope to address these issues with future projects.

**GUI:**

The main inspiration and theme we used for our GUI is from the game Undertale. Created by an indie developer, Toby Fox, Undertale is a role-playing game that is currently gaining massive popularity among the gaming community for its creative gameplay mechanics and player-choice drive plot. With some of our members having played the game and coming up with the idea of possibly being able to replicate and edit the original sprites used in the game, we decided to make it our theme. Even though the main concept of our GUI is Undertale, we did not rip anything, graphically, from the game. Our graphics and sprites were recreated, pixel by pixel, on Photoshop to mimic the look and feel of the original game. The board was also taken from the game, but recreated and edited to fit the dimensions of our project. We also edited the different rooms and the power-ups in the same way. Although we used the same sprite design, we had to manually make duplicates using Photoshop, including multiple frames for character animations, because the original sprites from the game were too big.

Our GUI is comprised of 3 different JFrame classes: the MainMenuFrame, the ManualFrame, and the GameFrame. The first of the three, the MainMenuFrame, is obviously in control of displaying the main menu to the user. Here the user can decide between starting a new game, loading a pre-existing game, opening the manual, or quitting out of the game. The background—one of the many images produced in Photoshop—is set by applying an Image object onto the content pane of the frame. The image, along with all other graphics for the GUI, is grouped together in the project’s “res” or ‘resources’ folder.

The second JFrame—the ManualFrame—displays an index of pictures that depict the functions of the game, from looking, to moving, to saving. The JFrame is composed of two components placed in the west and east sections of a BorderLayout. The west container contains a JList that acts as the manuals table of contents. Here, the user can click on whichever page of the manual he/she wishes to go to. By selecting any selection on the list, it will alter which image is displayed in the JLabel placed in the east section.

The last JFrame—the GameFrame—acts as the graphic-equivalent replacement for the UI. Rather that displaying the board to the player through the use of characters, the GameFrame displays the re-created pngs and animated gif sprites. The game is played in the same respect as the text-based version, except actions are now linked to KeyListeners. With each key that is pressed and/or released, a KeyEvent is generated in the appropriate method; within these methods are numerous if, else if, and else statements to act according the user’s input and reflect those actions in the interface—much like how the UI works. Movement is now controlled with the standard “WASD” set, and looking and shooting are controlled by holding “L” or “K” respectively, along with whichever direction of the “WASD” set the user wishes to act with.

The benefit of creating the GUI as the last portion of our project is that coding the class was as simple as replacing the methods from the UI with a javax.swing or java.awt equivalent action. Granted, it took a long amount of time to pick up a book, read multiple sections and chapters and GUI programming, and sampling test trials to reach the point that we did with this project. However, we were still able to finish this section of the code within the time frame, and with how much work we put into its completion, it’d suffice to say that we are the most proud of this feature of our project.

**Testing/Result:**

**Player Action: Looking**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Player in position [8][0]  Looked in Up direction  Enemy in position [6][0] | Enemy is located and result is displayed in console | Yes |
| 2 | Player in position [7][0]  Looked in Right Direction  Room in position [7][1] | Console prints that there is a room in the way | Yes |
| 3 | Player in position [8][0]  Looked in Down Direction  (Out of Bounds) | Console displays the there is a wall in the way | Yes |
| 4 | Player in position [8][0]  Enemy in position [7][0]  Shield in position [6][0]  Looked in Up Direction | Console displays there is a shield at that position | Yes/No |
| 5 | Player moved Up after looking | Spotted power-up returns to being invisible to the player, after the move | Yes |

**Player Action: Moving**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Player moves into null space | Player is relocated into null-space with new coordinates saved | Yes |
| 2 | Player moves into an ammo power-up | The player’s bullets counter increases by one and the player is moved | Yes |
| 3 | Player attempts to walk into room through various directions | When entering from above, the player can occupy the space; otherwise, console displays you cannot enter from the side | Yes |
| 4 | Player attempts to walk out of bounds | Console displays this is an invalid move and prompts for a new direction input | Yes |
| 5 | Player moves into room with briefcase | The console displays the end game message and starts a new level | Yes |
| 6 | Player moves onto an enemy | Console displays this is an invalid move, but makes the enemy invisible until a valid move is made | No  (Fixed in 2.0) |

**Player Action: Shoot**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Shot enemy from point blank range | Console prints and enemy was shot at the proper location; bullets decreased by 1 | Yes |
| 2 | Shot enemy from across the map | Console prints and enemy was shot at the proper location; bullets decreased by 1 | Yes |
| 3 | Shoot a wall (out of bounds space) | Console states the bullet hit a wall; bullets decrease by 1 | Yes |
| 4 | Shoot when out of bullets | Console displays the user cannot shoot when bullet is zero; asks for a different option for that turn | Yes |
| 5 | Shoot at a room with an enemy behind it | Console displays the bullet hit a room; decreases bullets by 1 | Yes |
| 6 | Shoot an enemy whilst in the room | Console displays the enemy has been hit; functions as if player is not in room | No |

**Game Actions: Saving Files**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Save any random game | Console displays that the game has been saved and the file exists in the proper directory | Yes |
| 2 | Save game with all power-ups collected | Console displays that the game has been saved and the file exists in the proper directory | Yes |
| 3 | Attempt to save game during Phase 2 of turn (after looking) | Console displays that this is an invalid move ( a player can only save during phase 1 of his/her move) | Yes |
| 4 | Save game in which all power-ups and enemies are gone | Console displays that the game has been saved and the file exists in the proper directory | Yes |
| 5 | Attempt to save while the player has lost all lives | Method for gameOver prevents the user from inputting any turns | Yes |
| 6 | Save game in debug mode | Console displays that the game has been saved and the file exists in the proper directory | Yes |

**Game Actions: Loading Files**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Loaded a saved game | Save is loaded and generated onto a new board and the game is executed from there | Yes |
| 2 | Loaded a game in which power ups were found (IE. The locator) | The save is loaded and the board is generated with the briefcase/goal room highlights, as intended | Yes |
| 3 | Loaded a file that does not exist | Console displays this is an invalid file and re-prompts the user for another file name input. | Yes |
| 4 | Loaded a file from a previous version of the game | Cannot load the file due to missing fields in the Data object; the Data file is thus obsolete in the newer version | Yes |
| 5 | Load a game that is in hard mode | Hard mode is not retained and must manually be set in game after loading | No |
| 6 | Load game in debug mode | Loads the game from debug mode, but cannot go back to original state | No |

**Game Actions: Move Enemies**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Set enemies to move after player moves | All enemies move properly | Yes |
| 2 | Set enemies to move when clustered in an area | Enemies can get stuck and thus crash the game if they have no valid space to move to | No  (Fixed) |
| 3 | Move player next to enemy to be killed | Enemy kills the player by taking over their space; player spawned back to initial position | Yes |
| 4 | Move player into room next to enemy | Enemy ignores the player next to him because he is in the room | Yes |
| 5 | Move enemies to corner player | If player cannot move, the game gets stuck and is unable to continue because the player has no valid move | No  (Fixed) |

**Hard Mode**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Changes mode to hard mode via the ‘H’/’Hard’ command | Console prints the game has been switched into hard mode | Yes |
| 2 | Player moves into the same row (left) of the enemy | Enemy proceed to move toward the player’s direction until line of sight is broken | Yes |
| 3 | Player moves in same column (down) of the enemy, but behind a room | Enemy moves right due to the randomly generate location | Yes |
| 4 | Player moves in the same column (up) of the enemy with a power up in front of the enemy’s path | Enemy moves right due to invalid collision with power-up | Yes |
| 5 | Player moves into a room that is the same row (left) of the enemy | Enemy moves left toward the player; however, the enemy is still unable to kill the player | No |
| 6 | Player gets Shield then runs toward the enemies | Enemy moves randomly as he still cannot kill the player | Yes |

**GUI**

|  |  |  |  |
| --- | --- | --- | --- |
| Test # | Test | Result | Worked |
| 1 | Ran method and inputted for a graphic interface | Opens the MainMenuFrame with the menu buttons | Yes |
| 2 | Pressed the [1] Start New Game button on the MainMenuFrame | Opens a new GameFrame with a new board and the loaded HUD and Text Box | Yes |
| 3 | Pressed the [2] Load File button on the MainMenuFrame | Opened the file chooser window and once a file was chosen, loaded said board into a game engine used to create the GameFrame | Yes |
| 4 | Pressed the [3] How-to-Play button on the MainMenuFrame | Opened the ManualFrame window with functional JList table of contents and proper sized tutorial images | Yes |
| 5 | Pressed the [5] Quit Button | Window is closed and program is terminated | Yes |
| 6 | Moved, Looked, and Shot in GameFrame | All functions work as they do in the text-based version, as expected | Yes |

**Conclusion:**

In conclusion, we were able to finish the project with all the requirements given to us including the extra credit of implementing a hard mode and a GUI. Within the three weeks we worked as a team creating, coding, and molding this project and during that process we were able to learn several lessons. Teamwork is necessary in the future and we were able to touch the beginnings of that concept as computer science majors in this project. It wasn’t easy having to balance our time accordingly to those of our teammates and managing the work between everyone. But by going through this process, we were able to learn and experience something new that will benefit all of us in the near future. It’s been a difficult three weeks working on this project, from all the coding to creating the GUI, but in the end we learned a valuable lesson of teamwork.