CS 141: Intro to Programming and Problem Solving
Professor Edwin Rodríguez

Final Project Spring 2017
The Ornithologists:
Eduardo Saenz (Team captain)
Jacob Chong
Lance Dall
Bumjoong Kim
Corey Perez
Grant Posner

Introduction

This project was designed to help students create simple turn-based game using objected oriented design philosophy extended from the Java object-oriented coding designs originally learned from the CS141 class.

The game design was to bring fun and challenge to the player who enjoys playing simple text-based adventure game. This would be achieved through an object-oriented design that allowed for turn-based commands from the players to be inputted to the game and have the game react accordingly.

The major objectives of this project are:

- Create a turn-based text game that will contain full functionality and a user interface that is able to be understood.

- Introduce multilayer functionality in the game that will allow several elements that comes with effect into play in a game simultaneously. It should be able to give experience to the player last to survive and to capture the briefcase and ninja to stop the player in real time game experience.

- Introduce computer controlled randomized allocation of ninja and items to make the game more challenging and interesting. The movement and action of all objects/entities in the game will be controlled by computer, with the exception of the player's own movement.

Problem Description

The player begins within a building, composed of 81 cells arranged in a 9x9 square. Within the building are 9 rooms, one of which contains a briefcase, which the player need to collect to win. The building, however, is pitch black, and filled with Ninjas out to stop and kill the player.

The player is equipped with Night Vision goggles and a Gun with one bullet. The player can see two cells in any direction, and, once per move, see if a ninja is in a chosen direction. The player can also shoot in any direction, killing any Ninja in the bullet's path.

After the player moves, the Ninjas will check their surroundings…
If the player is adjacent to a Ninja, the Ninja stabs the player, ending their current life, and resetting them to the entrance of the building. If the player is not located in all the direction nearby, the Ninja will move one cell in a random direction.

The player wins the game when the player enters into a room that is with briefcase inside, and the rooms on the grid are accessible to the player from the north side only and their content is not visible.

There are power-ups randomly displaced on the grid for the player to use for his survival. there are three types: bullet for ammunition, invincibility power-ups, and radar for visibility. Taking a bullet (noted as 'b') replenishes the player with ammo, but if the player already has ammo, then power-up does nothing. Taking invincibility power-up, (noted as 'i') keeps the player to be protected from stabbing for the next five turns, and taking radar power-up (noted as 'r') reveals location of the briefcase on the grid.

This project introduced a new challenge to the programming assignment. we as a team had to learn how to create game engine and load the game elements to the display and be able to save the game elements to a file and load the game status from saved file.

Approach to solution (Design, Architecture)

We split the group into three sub-teams, each designed to have a special emphasis on a certain piece of the game's design and development. These groups were Game Engine, User Interface, and Miscellaneous. The Game Engine's goal was to design and implement a modular game engine that would be a solution to the project's problem. Eventually, the Game Engine team also implemented a lot of logic within the Game Board, which was bloated with too much logic earlier in development. The User Interface team was tasked to design, write, and implement a textual user interface that would allow the player to interact with the game. The user interface became a center where all game engine actions were called from, and was also where the game's main loop would be run from, which proved temporarily effective, yet somewhat confusing to an outsider looking at the code. The Miscellania team would design and implement how all other objects within the game would function, the team was also given a secondary function as overseers for the project's whole, and would attempt to fix any discrepancies or bugs in code throughout the other two teams. This included things like all power ups, the player, the ninjas, and more. The Miscellaneous team would also initially implement the Game Board, and it's printing.

1. User Interface Team (Eduardo, Bumjoong)
2. Game Engine Team (Jacob, Grant)
3. Miscellaneous Team (Corey, Lance)

Programming the project had been pre-split into 4 main deadlines, or milestones, set by Project Manager Edwin Rodríguez. These milestones were set about one week apart from each other, and were as follows:

1. **Milestone 1:** Architectural design and skeletal code. **05/10/2017**
2. **Milestone 2:** Implementation of Grid complete. Program should be able to instantiate and initialize (random placement of ninjas/powerups/briefcase) the Grid, and print it. Code should have debug mode at this point so Grid initialization can be shown. **05/17/2017.**
3. **Milestone 3:** Game is playable. UI details don't have to be finished, but the game is playable in some form, e.g., start game and have raw commands for movement, etc. **05/24/2017.**
4. **Milestone 4:** Game is finished, including saving and loading. **Presentation Day.**

During the week of Milestone 1, all teams met and discussed the overall design of the project via Discord. We soon decided on the main classes, (Game Engine, User Interface, Player, etc.), and their functions, the Miscellania team also created classes for all the game's other elements. Skeleton code was then created for each team by their respective class. At this point, all code was still being shared via zip files over Discord. By the milestone's end, we had the following classes in place with Skeleton Code:

1. ActiveAgents
2. GameBoard
3. GameEngine
4. UserInterface
5. Gun
6. Briefcase
7. Bullet
8. Knife
9. Invincibility
10. Mainlaunch
11. NightVision
12. Ninja
13. Player
14. PowerUps
15. Radar
16. Rooms
17. SaveState

Milestone 2 set many fundamental fields and classes for the project. All entities that would be on the Game Board were given location fields and methods for getting/setting these fields, also, (by the Project Manager's advice) unnecessary classes were removed from the design of the project (these included NightVision and Knife). At this point, each of us had learned to use a git manager and were collaborating on code in an organized fashion via BitBucket and SourceTree. During the milestone, several things were added:

1. A barebones UI class, which allowed for navigation in one direction, and choosing of a Graphical UI or Text UI (though no Graphic UI was ever implemented).
2. A GameBoard class, which would create all entities on the gameboard (ninjas, player, powerups, rooms, etc.) one by one, calculate their positions, and then print a board based on the grid's current contents and the rules of the game.
3. A GameEngine class, which only had methods to create a GameBoard and run the gameboard's print methods.
4. Each Game Board entity was given row and column fields.
5. Ninjas were given a calculateRow() and calculateColumn() method which randomly calculated a position for them.
6. The Player class created a Gun object and had shoot methods

Milestone 3 was a small overhaul from the previous iteration of the project. At this time, the GameBoard class was larger and held almost all of the game's logic and the game's objects. This made it essentially the De Facto Game Engine, with the actual GE only holding calls to the GameBoard and a few methods to set initial player/briefcase locations. This was solved by moving the few methods from the Game Engine to the Game Board, and renaming the Game Board to GameEngine. Then, a new GameBoard class was created with a new printing method. The grid

would now be printed by comparing the locations of entities on it. Some would take priority over others so two objects can have the same position coordinates, but when in printing, only one would print. This allowed for ninjas to step over powerups without ‘eating’ them, since the ninja would print when they took the same space, but once the ninja leaves, the powerup will print once again without a problem. The object printing priority:

1. Briefcase
2. Rooms
3. Player
4. Ninjas
5. Powerups

At this stage, the three development teams also began to mesh together and fade, as the project’s scope became larger, and the classes depended more and more on each other to continue development, some teams would implement methods on classes outside their range to allow for a certain implementation to be created, and as more teams did this more often, team lines began to blur, and the project became one large team.

Also, the UI began to take a better shape, as a debug option was implemented, and placeholder about/loading menus were added, with an option to go back to the previous menu. Simple collision detection between player/ninjas and rooms would stop them from going into rooms, and made the game feel just a little more complete.

Milestone 4 brings the project to what is essentially it’s current state. As the project came to near completion, the saveState, testMain, and ActiveAgents classes were removed, and several new grid implementations were created. These included looking, the player’s view, shooting, a pause menu, saving/loading, player lives, ammo, Ninjas can now stab the player, powerup pickups, respawning, and more. More information about how these systems work can be read inside of this report’s **Implementation** section.

Things in the final implementation that have not yet been created/fixed:

1. Ninjas won't stab player if the player walks into the ninja’s space.
2. Player could see through the walls of a room
3. Player is not stabbed when checking for a briefcase, and a ninja happens to next to them at that time.

For the sharing ideas and creating tasks, we used Discord, a free chatting and VOIP service, as a communication channel within our group. Meetings on Discord would be arranged to talk about the project’s outline and what implementations are left to complete. The ideas discussed in the meetings would become fundamental railings that guided us to the project’s end. Discord also allowed for us to share zip files of our code, before we implemented a git solution to better organize the project. Our team used Sourcetree as this git solution, which fed on the code updates and shares edit progress on the repository and the different versions branch out where the fixes are. The sourcetree proved as a very useful control management tool; code was easily accessible and it was easy to locate a point in our programming cycle because it keeps each updated versions of the progress in each part when the program edits occur. Code changes were easy to understand, since direct changes were shown and color-coded, and committers were required to write a commit message (some of which were more detailed than others). Most of us decided to use Eclipse as our IDE since it was prompted to have user-friendly setups and convenient git implementation, which would show changes and the current commit.

Discussion of Implementation

This section will discuss the several systems that are in place in the final iteration of the project.

a. User Interface
    i. States

State and previousState are two fields which are essential to the function of the UI. The entirety of the User Interface is broken up into Nine main ‘states’ or pages that the game will be in. Each state will represent a different menu or game state that the player can be in at any time, and they are run within a state switch nested in a while loop. Each state will eventually change to another state using the changeState() method. The changeState() Method takes one input to be the next state, and changes previousState and State accordingly. The 9 game States work as follows

1. The Game’s first menu. Here, the player will be prompted to choose “New Game”, “Load Game”, “About” and “Exit”. These options will Create a New Game, go to the load menu, go to the about menu, and exit the game, respectively.
2. The about menu. Here the player is given information about the game and it’s controls. The player can return to their previous state by pressing ENTER.
3. The loading menu. Here the player can enter a save file to load. they can also return to their previous state.
4. the UI menu. Here the player can choose to run a Graphical UI or a Text-Based UI. The Graphical UI was never implemented, and thus the option currently tells the player it is under construction.
5. This state is the only one that is not an actual menu, but a state of the Game. Here, the Game Engine is initialized either by creating a new one if the player selected “New Game” or setting the loaded Game Engine to be used for gameplay if the player selected “Load Game”. Then, depending on the player’s choice for UI, the respective UI is loaded. The main game loop will run here until the player pauses, exits, or wins/loses.
6. The debug menu. Here the player can choose to run the debug mode of the game, which shows all spaces and the briefcase location from the beginning.
7. The Pause menu. Here the player can load or save a game, look at the controls, exit to main menu, or exit the program. Player can also return to the game by pressing P.
8. The Saving menu. Here the player can save their game by choosing a save file name. Names must be greater than 3 characters.
9. The Game Over Screen. Here, the player will see a screen that will tell them if they won or lost the game, depending on their actions within the game.

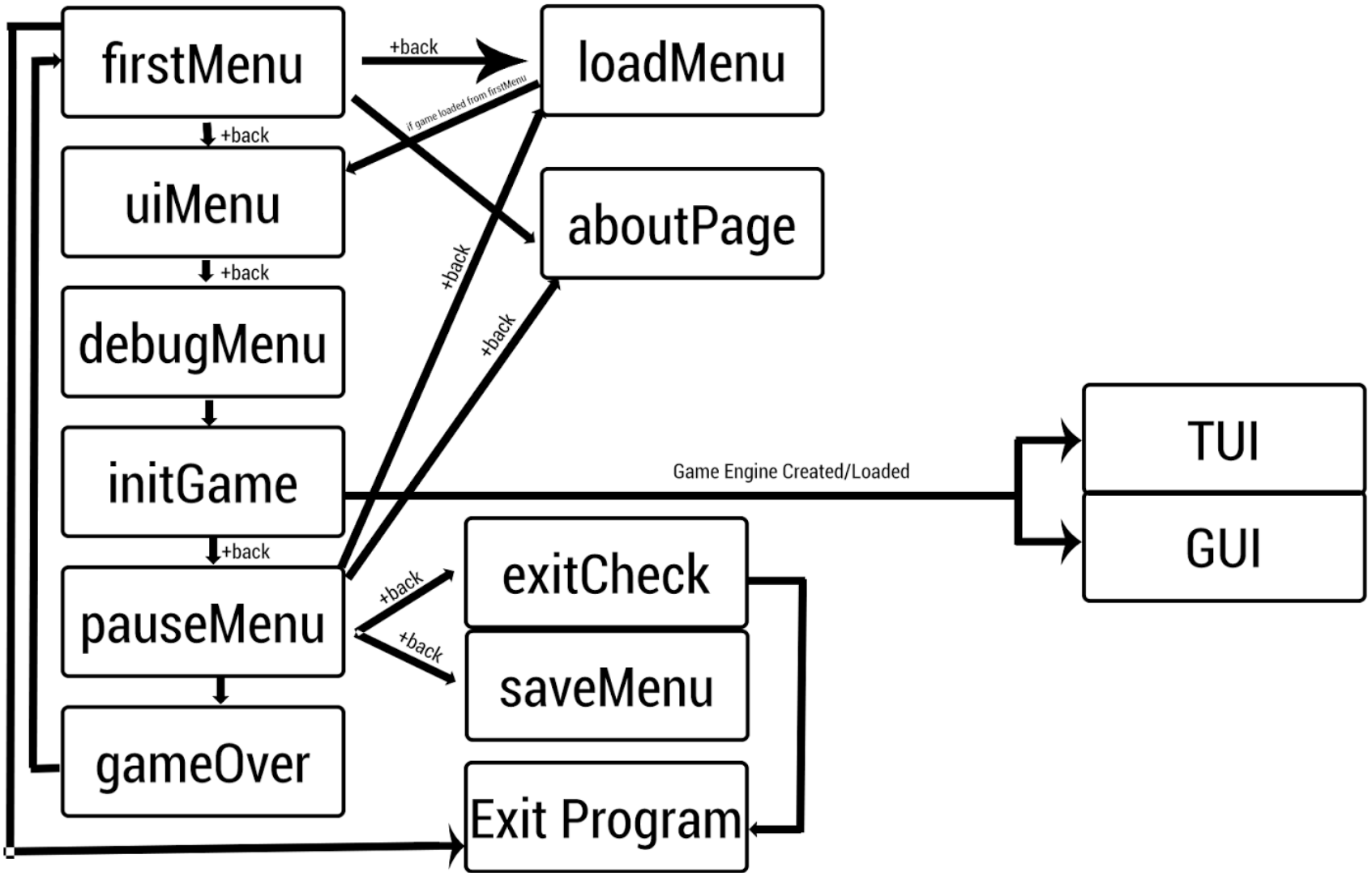
See the page 8 to view how the UI state loop runs more in depth.

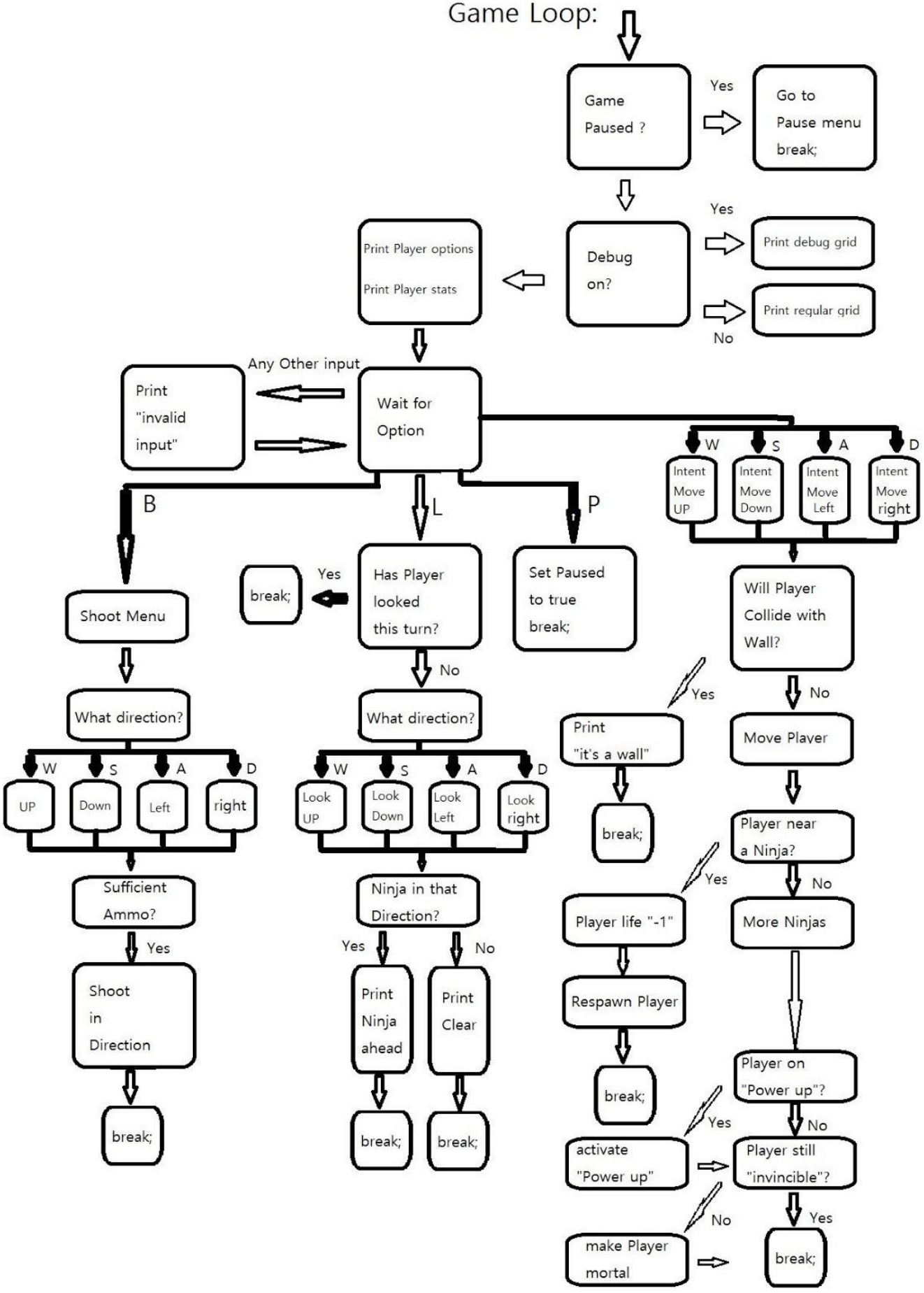
ii. Game Loop

Located in State 5, the ‘game loop’ is run initially when the game begins and again every time the player moves. It stops looping when the player wins, exits, or the program stops. The game loop performs several actions to keep the game functioning properly. The Game loop follows this procedure:

1. Calling the GE to see if a game-over scenario has been reached. If so, set the state to show gameover screen.
2. Calling the GE to print either a debug or regular grid, depending on player’s choice earlier.
3. Displaying the player’s stats and current options.
4. Waiting for a new player input.
    a. Calling other UI methods to react to non-moving inputs like shooting or looking, then calling the GE to react to these actions.
    b. Calling the GE to move the player if a move input is entered, make ninjas check for spies, move the ninjas, check if the player is on a powerup, and check if the player still has invincibility.
    c. Printing ‘Invalid Move!” when given an option that isn’t programmed.

A more detailed look at the Game Loop can be seen on Page 9.


firstMenu
+back
loadMenu
if game loaded from firstMenu
+back
uiMenu
aboutPage
+back
debugMenu
+back
+back
initGame
Game Engine Created/Loaded
TUI
GUI
+back
pauseMenu
+back
+back
exitCheck
saveMenu
gameOver
Exit Program


Game Loop:
Game Paused ?
Yes
Go to Pause menu break;
Debug on?
Yes
Print debug grid
No
Print regular grid
Print Player options
Print Player stats
Any Other input
Print "invalid input"
Wait for Option
W
S
A
D
Intent Move UP
Intent Move Down
Intent Move Left
Intent Move right
B
L
P
Shoot Menu
break;
Yes
Has Player looked this turn?
Set Paused to true break;
Will Player Collide with Wall?
What direction?
No
What direction?
Yes
Print "it's a wall"
No
Move Player
W
S
A
D
UP
Down
Left
right
Look UP
Look Down
Look Left
Look right
break;
Player near a Ninja?
Sufficient Ammo?
Ninja in that Direction?
Yes
Player life "-1"
Yes
No
More Ninjas
Yes
No
Shoot in Direction
Yes
Print Ninja ahead
Print Clear
Respawn Player
Player on "Power up"?
break;
break;
break;
break;
Yes
No
activate "Power up"
Player still "invincible"?
No
Yes
make Player mortal
break;

b. Game Engine

i. Initializing a new Game Engine

a. When a new GameEngine is created, objects for all entities on the board are created, and then their positions are all calculated. This includes the Rooms, Ninjas, Player, Briefcase, and Powerups.The GameEngine holds all these created objects, as well as getters for their positions. These calculations are used in the printing of the board, as well as other things like collision or looking.

b. The GameEngine holds methods for each ingame action that can be done by the player, and any other entity on the board. This includes player looking and shooting, ninjas stabbing, and powerups being picked up as well as other actions. The more complex methods are explained here.

i. Active Looking

An Active look can be done once per turn, and is begun by entering “l” or “L” when it is the player’s turn. Then, any WASD direction can be entered to look. Depending on the direction, the game engine will then check every row/column that is before/after the player’s current location, and check for a ninja. If any of those spaces have one, then a “ninja ahead’ signal is printed, otherwise a ‘clear’ signal is given.

ii. Shooting

Shooting is a lot like looking. Once activated with “b” or “B” and given a WASD direction, the Game Engine will again check if the column/row before/after the player’s location. The first Ninja it runs into has it’s ‘alive’ parameter set to false, and the game carries on. No signals are given with shooting.

iii. Collision

Each collision method works in essentially the same way. The methods each take one intended direction as an input (in the form of a String such as “up”), and based on that, the cell one row/column before/after the colliding object is compared to the cell that any given colliding object is in. If any is the same, then the method returns true. If not, then it returns false.

iv. Game Saves

With Game Saves, the entire Game Engine is saved to a file. Since the Game Engine contains each object and thus its position, the game’s state as a whole is saved to the file. Loaded Game Engines are applied to the UI object, which will result in the Game’s state being loaded.

v. Ninjas Stabbing and Player Dying

After the player moves, Ninjas will check every adjacent cell and their own cell for the player. If the player isn’t on one of these cells, then nothing happens. If the player IS on one of these cells, then the player is stabbed via stabSpy(). stabSpy() will check if the player is invincible currently. If not, then the player’s lives are decremented, a “you were stabbed” prompt is given, and the player’s location is set to the starting point.

vi. Collecting the Briefcase

Every time the player moves down, a check is done to see if the player is above a room. If so, the method checkRoom() is run. checkRoom will run through each room: if the player is above it, and if it has a briefcase a prompt telling the player they found the briefcase is shown, and the game over/win fields are set to true, ending the game.

c. The Game Board

i. Printing the board

a. Debug

The debug board printing process follows a series of steps to print all board entities with correct placement and layering. Some entities will take priority, as they are more important for the player to know about than others. The most notable instance of this is with Ninjas taking priority over powerups. This layering is done by building the board cell by cell, in an ordered process:

i. The Cell is set to be empty (" ")
ii. If the Cell matches the player's location, it is marked. ("P")
iii. If the Cell matches a Ninja's location AND is empty, it is marked ("N", 6 total)
iv. If the Cell matches a location of one of the rooms, it is marked. ("R", 9 total)
v. If the Cell matches the Briefcase location, it is marked ("B")
vi. If the cell matches a powerup location, AND is not yet marked, it is marked accordingly. Thus, only if the powerup does not share a location with another entity can it be printed. Powerups all print in this same way, in this order:
    1. Bullet ("b")
    2. Radar("r")
    3. Invincibility("i")
vii. Then, the cell is printed to System.out.

As a result, all cells are printed with the correct building layout.

b. Regular

The Regular board printing process is very similar to the Debug process, with a key difference: Only certain cells are set to be empty. With our game, we added a "Passive Looking" feature to allow the player to see a set amount spaces in all directions (by default, 2), without a look action. A check is done to prevent the player from seeing through rooms. The Regular print process is as follows for each cell:

i. The cell is set to be invisible ("*")
ii. If the cell is within the player's viewdistance, it is eligible to be visible, and a check is done. All cells between the eligible cell and the player are checked for rooms. If a room is present, then the player cannot see that cell and it loses its eligibility. If no room is present between the player and the eligible cell, then it becomes marked visible/empty (" "). Note this check is done 4 times, for each direction.
iii. The same Debug print process as above is run from here, sans the initial empty setting.

As a result, a grid is printed with only a specific sample of cells showing the real board, and the rest as invisible cells ("*").

Testing data

Testing our code was another essential part of our development cycle. Various bugs were found that needed to be fixed, and tiny nitpicks from how the game ran to how our group's work went would lead to quality-of-life features we implemented to the final design.

Bugs or Problems Found:

a. Ninjas erasing marks for powerups
   At the second milestone, ninjas would be able to walk over power ups, but the powerup mark would not return when the ninja eventually walked off of it. This was a result of the older grid printing algorithm, which didn't support printing fresh, and based prints off the previous one. This was fixed in the printing overhaul that occurred by milestone 3.

b. Unable to look to the right in bottom row
   When looking was first implemented, a very specific bug occurred with a beta version of the passive looking. This version of looking was like the passive looking in the final version, but required input, and only went one direction revealing two spaces in that direction. This version was not able to look right when the player was on the first column, and no one could figure out the problem. Eventually, this problem lead the team to scrap this looking system and lead to the split active and passive looking system currently in place.|

c. Certain gamesave data corrupted.
   When implementing Game Saving and Loading, some initial test save files were created that loaded empty game data. Most of these files had no effect, but one had a permanent effect on the game that didn't allow a new Game Engine to be created or loaded for that session if the save was loaded.

Suggestions for Improvements, Lessons Learned

All projects can be improved, but maybe some more than others. Ours is no exception to this, and is far from perfect. Seeking for new improvements is important to drive developments, but it's also important not to be too critical of one's work, and set goals to be too high. This project also helped us each learn a lot about the development cycle and working as a group on code.

a. Code is somewhat messy
   Collaborating on code with 6 people each with a coding style would obviously lead to some code blocks looking very different from others. Trying to understand the code as a whole when each code block has a unique path of logic and syntax style proved to be more difficult than it sounds. Comments also played a large part in understanding code, and as the project became more complex, creating comments to help follow the logical path would become a chore, and eventually were left out, leaving large blocks of complicated, uncommented code floating about each class, leaving everyone but the author confused about its purpose.

b. No class to extend other classes
   When the project was in its infancy, we placed many superclasses that would extend fields and methods into others. These classes, like ActiveAgents, were scrapped since their to-be children were too unique to be contained within the superclass. Later, the printing overhaul resulted in all gameboard entities to have a column and row field, as well as getters for those fields. These classes could have been contained in a "GameEntity" superclass, to simplify the classes.

c. Procrastination/Sticking to Teams
   Procrastination was a formidable opponent for the group, and a large reason that our initial dev team would break down by the end. When one team fell behind, other team's couldn't progress forward, since essential code to do so was missing. This created several problems that could have been avoided with better planning and organization.

Conclusions

We learned a lot through this project. Difficult obstacles came from both from the problem given and from our path to a solution. Given how it turned out, it should be fair to say we did well. The game solves the problem, and does so with reasonable solutions, but the game isn't without faults. Improvements can be made and nitpicks can be fixed, but the project's problem was solved, and done so with reasonable logic.

Working on a game from scratch wasn't a new problem for us, but doing so with a group, and in a grand scale, definitely was. Adopting ideas and information off of each other and online resources helped us push our way toward the answer, and having others to help with a small problem was very helpful and effective. However, having others work on code also lead to problems. Discrepancies within what solution to use and even how it should be implemented arose, but were conquered. In the end, we learned how to collaborate effectively, and build something together that we couldn't have done alone, and there's something to be proud of in that.