Phase 2 Report – an analysis of *Hidden Squirrel: Peanuts and Acorns*

Describe overall approach to implementing game:

Implement a menu where a button will be used to initialize a new game in the following structure:

Starting from the backend, we used a 2D array 25x15 of enumeration objections to represent the information at any location at every game tick.

Initially, the board will be randomly generated with walls, traps, rewards, and enemies. At every tick of the game, the board will be manipulated based on the hero's movement (using KeyListener methods from java libraries to get user keyboard input). Each game tick effects the following elements on the board ->

Enemies position : shortest path to the player

Entities updating : pick up bonus, regular rewards, traps,

Hero Position :

Once the board gets updated, we can use the java library AWT to generate a new image on the screen. Every object in the array associates with a 60x60 pixel to be generated on the screen. Collisions would simply be detected if two objects exist on top of one another, and this would indicate most of the use cases for the game ex: lose if enemy touches player or win if player is on the exit door tile. Moreover, running a depth first search is now possible using this 2D array implementation. Conclusively, every tick the board updated, an associated image gets drawn on the screen based on the objects within the board.

As the game grew more complex, we realized that rendering should not be as simple as making a new image every tick. The movement seemed very rigid where characters would move from one tile to the next (60x60 pixels) which made the jumps seem too big. As a result, there was a happy medium where images drawn were broken down into 7 different frames per tick to improve smooth game mechanics (see java/Display/myGame).

Moreover, new displays (type Jpanel) needed to be created where buttons within these displays allowed navigation into other features. In the end, the game grew to not only the actual playable part, but as well as a useful settings menu, sound features, difficulty selection, game won/over screens.

Class diagrams

New: Display became a class of 7 different Jpanel extended classes which merged previous old Display/MenuLogic/Game classes

Package: Display -> Title, Play, Settings, Difficulty, Pause, Game Won, Game Lost.

DisplayLayout contained a variable displayPanel which contained all of the 7 renderings as well as the startThread() and run() methods simulated the game loop. It also included the rest of the other classes such as DataStructures, GameLogic, BoardData (which were all replaced in the new UML model)

MenuLogic was basically replaced by the Settings/Title classes which included many more renderings and buttons

Game, it was in a JPanel containing functions to update() repaint() the game based on board/object data and player movement. A lot more variables were needed for JPanels, to store images, JButtons, JLabels.

Game Won, Game Lost, Pause included some art for the background a few buttons for going back to the main or for restarting the game, see use cases.

New: Entities_pacakge -> following class changes/implementation

StataticEntity/MovingEntity : extend class position for {x,y} coordinate getters and setters

Exit class created for the door added a handful of functionalities needed for game development,

Bonus class overhaul for Spawntime and Despawntime variables/methods for making the rewards disappear.

New: Gamelogic class -> Game package : following classes implemented

HeroLogic : methods for processing player movement and private methods for reward/trap collision

EnemyLogic: methods processing enemy movements toward enemies and collision reward/trap collision (different from hero detectio of course

RewardLogic: methods updating bonus rewards which involved despawning/respawning them in ifferent locations

New: BoardData class -> Board package: following classes implemented

BoardData: same idea but added a ton of private methods for random terrain generation

Difficulty/Objects: enumaration of difficulties and possible objects for the 2D array

New: Helper package: used for globally used information not pertaining to any specific class

Direction: NORTH,EAST,SOUTH,WEST enumeration for map orientation

HeroColor: choosing hero skin

KeyHandler: Takes player movement using java awt method overides for KeyHandler(), keyTyped(), keyPressed(), KeyReleased()

Node: Node constructor used for DFS on enemy pathing

<u>Use cases updates:</u>

Main class would make the JFrame executable window with all the screens available, and then divides some work for the game loop onto new thread so the main thread is able to do other tasks.

Starting the game consisted of pressing a Jbutton to begin a thread inside of myTitle, which would invoke run() method to program inside a while loop until an ending condition was met (see Phase 1 use cases). Every tick (525ms), the run() method would run an update() method to invoke hero,enemy,reward logic which would effect boardata inside of Object data class. Then, repaint() is invoked to show the current state of the game on the screen based on the boardata and the previous update() invocation.

Jbuttons use case for navigating to different features of the game. Each button had an associated actionListener(event) and actionPerormed(event) methods to which would be used to navigate the above listed displayed (as well as overriden paintComponent(Graphics) for showing the display.

Initial designs grouped up every kind of settings, starting game, and exiting program into this class, but they can be separated, where settings are in settings, starting game would be used from clicking a Play Button in the Title screen, and Exiting program also can be in the title screen.

For playing the game, it had a separate difficulty selection in the main menu instead of after pressing Play from the Title screen. This is to let players have the same level whenever they hit the play button and reduce the number of times a player would need to see the difficulty menu. It lets a nice transition from Play button into the game, instead of Play button, pause to check difficulty, then play game.

No restart option was added during the game won/over and pause screens. The game starts after the first direction or escape key press, so there is enough time beforehand to plan the route to take, making it easy to clear even in hard mode. Optimistically, the player would choose and complete the easy, medium, and hard difficulties with a couple of attempts, then the player is finished with the game.

Management process, division work:

In the first week, scrums would consist of members describing what part of the game development peaked interest. Since all group members were strangers to one another, no one knew the strengths or weaknesses of one another. Instead, we divided labor based on the interests of each person. During the second week, bidiurnal scrums would consist of the members discussing their progress in the following areas:

Sadaf: Specializing in the creativity and graphics were not only her interest, but also her strong suit. She took responsibility for most of the art inside the game. As the game progressed, she began implementing sounds to sharpen the game's overall appearance and feel.

Gabriel: Took on the role of front-end development for rendering the games various capabilities such as: main menu, settings menu, game over screen, playing the game etc. He became very familiar with java swing and awt libraries to work with different displays (Jpanels), buttons, and rendering images onto the window using threads for tick rates.

Misha: Specialized in interactions between objects within the backend. The logic/interactions between moving entities (hero/enemies) and static entities (rewards/traps) including picking up rewards, enemy collisions, win and loss conditions, etc. were done by Misha. He also implemented enemy movement AI using a breadth-first search to the path towards the hero, regular-reward and enemy random generation, and handled the animation and most graphics for the non-menu portion of the game.

Kevin: Specialized in board generation. Random walls, bonus rewards, traps, and generations were implemented to drastically improve the games replay ability. This type of map generation also allows for different difficulties when manipulating the number of rewards, traps and enemies to generate. Furthermore, the bonus reward mechanics were drastically enhanced since they would respawn and respawn at different rates.

Towards the final week, the big sections for style, front-end, back-end were wrapping up but many small changes in different areas needed to be fixed. As a result, all members could choose parts of other peoples work to develop (I.e Misha provided a huge rendering overhaul for entity movement(animation) which was initially under Gabriel's authority) and notified the group members in the discord group chat.

Enhance Quality of code:

1. Strict variable, method, class naming style: Camel case.

2. Most attributes were set to either private or protected. Any class/object which needed inaccessible data would use getters and setters.

3. Added folder directories associated to different packages and subtasks of the program such that finding where finding files for changing code becomes a lot more organized :

- Board, Display, Entities, Game, Helpers, Logic

4. Restricted all methods lengths to at most 75 lines of code. Any code longer should be broken up into smaller portions using private methods. Notably:

- implementing new classes for the 7 different possible displays which were initially all initialized inside DisplayLayout :

  Difficulty, myGame, gameOver, GameWon, myPause ,mySettings, myTitle

- seperating hero updates, enemy updates, and entity updates into different subclasses

  EnemyLogic, HeroLogic, RewardLogic

  -> which intern had multiple private methods for detecting games updates

  processPlayermovement(), processEnemyMovement(), updateRewards()

- separate factory class for generating objects : ObjectData

- separate factory methods for getting image PNGs inside of myGame : getImages()

External libraries used:

The two main libraries used were java swing and java AWT. Both contributed equally important roles for both the front-end and the back end of the game.

Java Swing laid the foundation for different displays, using components such as JFrame, JPanels, JLabels, and JButtons. Having the JFrame be an executable window for the game, different displays being the JPanels from Java Swing were necessary, used to act as a container to hold the many different components on it, such as the background image, buttons, and labels. Instead of constantly updating drawings to simulate selecting options on every display, the Jbuttons were useful to set up change screens buttons, setting up selected options to be pressed, and there are layout managers to organize the placement of components. When games are played, the drawing component in the JPanel helps to show the game playing by repainting multiple times to replicate frames per second, as well as visualize the game.

Java AWT helps manage after setting up the display and buttons, where this library is used to communicate with what happens when JButton is pressed and create/modify file images. When buttons are pressed, there should be some changes happening to the window, such as switching displays or changing game settings, and the ActionListener in AWT lets us implement different blocks of code executed for each specific button. For the images, they were in a resource folder, and by using BufferedImage from AWT, image data was able to be stored in the buffer, used to draw onto a JPanel as the background or drawing constantly onto a JPanel to show animations.

Biggest challenges:

Familiarizing with GitHub/IntelliJ proved to be a large problem in the project's development

Intellij IDEA issues:

> Needing to identify files as source roots/ resource roots when dealing when fetch data from repository. Many errors were caused by project structure resetting from git pulls. A common error was to continuously specify the Module SDK. Moreover, many problems often could not simply be resolved from print statements and required members to use the built-in debugger; a much better practice than using System.out!

Version Control Merge Conflicts:

> Two incidents occurred where a simply merge overwrote a lot of our game's logic due to a conflict between the local repository and the remote repository. As a result, production was down for a few hours at a time before the group could continue working with the head of the branch. Fortunately, this taught members how to branch off certain commits ID (to continue working), undo recent pushes, and solve *detached head errors.*

Random map generation:

During map generation we faced some issues with a map generation getting stuck in a loop, this was caused by two constraints. The first was that tree “segments” do not touch other segments; this was to avoid generating a location on that map that is entirely inaccessible. The second constraint was regarding a minimum distance between rewards, this caused issues when the reward count and minimum distance were too high. The minimum distance problem was addressed by playing laying a limit on the number of rewards. The tree problem required a counter for ending terrain generation when too many failed generations occurred. We decided the problem was fixed when no infinite loops occurred after generating the map difficulty on hard (the most entities) 1 million times.