Nilesh Gupta

604489201

CS 32

**Project 3 Report**

**StudentWorld:**

I made the StudentWorld constructor assign the value false to the level complete boolean since this is the only variable which should have the same value throughout the whole existence of StudentWorld (even when the player dies). The StudentWorld destructor just calls the cleanUp() function.

The virtual init() function, inherited from GameWorld, sets the time to 1500, sets the current sub level to 0, and initializes my vector of vectors of Actor pointers by loading the level. Each vector in the vector represents one sub level and is given all the actors of a certain sub level. A player vector receives pointers to all the players in the sub levels. It lastly sets the current sub level to 0 since the player will always start on the main level when a level loads.

My move() function resembles the pseudocode given in the spec. First, it sets the display text at the top of the screen, it then calls the doSomething() method for each of the actors still alive in the current sub level. If the player dies after one of the actors moves, then the function returns that the player died and decrements his lives. If the player finished the level, then it returns that. It then erases died actors from the game, decrements the time, and checks again if the player died or if the level was completed.

The cleanUp() function deletes every actor from the vector, clears the vector, and then clears the player vector. I was having trouble with this initially since I was deleting every pointer in the player vector as well which led to bad access bugs.

The getCurrentSubLevel() returns the value of the private variable representing the current sub level (sub).

The isObstacle(int x, int y) method was needed to limit movements of the player and the robots in the game. It cycles through the actors in the vector for the current sub. If the actor’s location matches the location passed into the function, it checks if it is an obstacle and returns appropriately.

The anyOtherActors(int x, int y) function returns a vector of the actors at the location passed into the function. This was necessary for the implementation of bullet since I needed to check if the bullet had hit anything and to make sure if it had hit a square with two actors, if one of them could be damaged, it would be.

The exitCanAppear() function just checks if all the jewels and hostages have been collected, and if so, then it returns true; the exit object then uses this to set itself visible.

The levelCompleted(bool b) is accessed only by the world’s exit object when all jewels and hostages have been collected, and the player is at the same location as the exit.

The transport() function is called whenever the player steps on a gate. The current sub level value is changed to the gateNumber of the gate.

I wrote the addActor() function so that whenever an object was created during gameplay, it could be given to the StudentWorld to manage.

The setDisplayText() function is called in the move function every tick and updates the string shown at the top of the screen. The statTextFormat() just formats the string shown using the functions show in the string stream page.

I had to write the functions gotHealthGoodie() and gotAmmoGoodie() and not for the extra life goodie since the extra life goodie just uses the incLives() function in gameWorld. The health and ammo goodies have to modify the player actors across all sub levels which can only be done in StudentWorld.

I wrote two separate functions, isPlayerLoc(int x, int y) and getPlayerLoc(int x,int y), since I needed to use the former for the exit variable and goodies but I realized I would have to use the latter to implement SnarlBots and their shooting habits correctly.

The clearShot() function forces the SnarlBot to shoot only when it sees the player in a direct line of fire, unobstructed by obstacles. It gets passed the robot location and direction as well as the player location and checks all coordinates in between using the isObstacle() function.

The numberOfKleptos(int x, int y) gets called by the Klepto Factory to make sure the number of kleptos in a 6x6 square around the factory is less than 3 before it creates another. This simply checks all the locations in the square around the location passed. If the location is out of bounds, it breaks. If not, then it uses the KleptoAt(int x, int y) function which cycles through all the actors in the current sub level and checks if: 1. an actor is a kleptobot and 2. if it is on the location passed into the function. I feel like the numberOfKleptos() function could be optimized in some way to use recursion but I didn’t have enough time in the project to figure that out.

**Actor:**

I gave the Actor class a constructor that accounted for every value that an Actor would need to have assigned at its inception and passed all values that I could to the GraphObject constructor. I gave Actor a StudentWorld\* variable and a getWorld() function since all the actors would need to be able to access the world they were currently in.

I made doSomething(), isDead(), and damage() virtual functions since all Actors need to have these functions defined (they are called many times in StudentWorld) and most of the Actors do something different, and most Actors respond to bullets frequently. The isDead() function for many Actors simply returns hp == 0, but since some Actors were not given a private hp variable, I needed to define this differently for some classes and had to make it virtual.

**Player:**

I gave the player and every other Actor that wasn’t constantly alive an hp variable and made the isDead() function return whether or not their hp was 0.

For the players doSomething() function, I first checked to make sure the player wasn’t dead. I then got the user input for the player using the getKey() function in GameController(). If it was a movement key, I checked if the square they were trying to move to was an object. If it wasn’t, then they moved to that location and direction was changed appropriately. If the user pressed space, then as long as the player had ammo, the game created a new bullet object placed one space in front of the player depending on which direction the player faced. If the player pressed escape, then I set hp = 0 to kill the player.

I defined two functions for the player, the getHp() (returns the hp in % form) and getAmmo() (returns current ammo) functions, solely for the game stat text at the top of the screen. I defined two more functions, setHp() and setAmmo(), to implement the goodies correctly.

**Gate:**

I gave gate class a gate number as a private variable initialized when it is constructed. It’s doSomething() method, if the gate is not dead, checks if the player is at the same location as the gate. If so, the player is transported to the sub level corresponding to the gate number and the gate is set to dead.

**Jewel, Hostage:**

The Jewel and Hostage classes work similarly. Their doSomething() functions check if they’re dead. If not, then they check if the player is at their location and if so, then they are set to dead. In the case of the Jewel, the player receives 100 points.

**Exit:**

The Exit is the only Actor that is not set visible during its construction (the studentworld sets the exit visible in its move() function when all the jewels and hostages have been collected). In its doSomething() method, it checks if the player is at its location AND if it is visible. If both of these conditions are met, it accesses the studentWorld’s levelCompleted(bool b) function and sets the levelComplete variable to true. This alerts the StudentWorld in its move() function that the player has completed the level.

**Goodies:**

The goodies work pretty straightforward. Similar to the jewels and hostage and exit, they check if the player is at the same location. If so, the player is granted an extra life or restored health or given ammo depending on the goodie and the goodie sets itself to dead.

**Robots:**

I was initially thinking I should write a robot class that extends actor and that would be the base class for all the robots in the game. This might be a more efficient way to go about the implementation of the robots, but I decided to play it safe and just write the robots to extend Actor.

I gave each robot a ticks and a ticksCount variable. The ticks variable was set to how frequently they could move while the ticksCount variable equaled the number of ticks they had been alive.

**SnarlBot:**

The doSomething() method for the snarlBot checks if it can move according to its ticksCount and its assigned number of ticks. It then retrieves the player location and checks if the player is in its (unobstructed) line of fire. If so, it fires. If not, then it moves one space in the direction it is facing. If that space is an obstacle, it turns 180 degrees.

The damage() function for the snarlbot lowers its hp by 2. If its hp is below or equal to 0, then the snarlbot is set to dead and the player receives 200 points.

**KleptoBot:**

The KleptoBot has, in addition to the ticks and ticksAlive variables, a distanceBeforeTurning int, a distanceMoved int, and an goodieID int.

The doSomething() method for the KleptoBot first checks if it is dead and if the number of its ticksAlive warrants movement according to its ticks value. If these conditions are met, then it checks if shares a square with a goodie. If it does, and it does not already have a goodie, and the 1 in 5 condition is met, then it “eats” the goodie (deletes it) and returns. If not, and if its distance moved has not exceeded the distance it can move before turning, then it attempts to move one space in its direction.

Else, if its distance moved has exceeded the distance it can move before turning, or it has encountered an obstacle, it picks a random direction and tries to move in that direction. If there is an obstacle blocking its path, then it tries to move in each of the 3 other directions. If every direction is obstructed, then it sets its direction is the random direction it picked and it does nothing else.

The damage() function decreases the kleptobots hp by 2. If it is dead and it was carrying a goodie, then the a new goodie is created of the same type at the location the klepto died.

**KleptoBotFactory:**

The Factory’s doSomething() method checks if there are less than 3 KleptoBots in the 6x6 square surrounding it, if there is no Kleptobot currently sharing the same square, and if the 1 in 50 condition is met. If all three conditions are met, then a new KleptoBot is born at the same square the Factory is at.

**Functionality Fails/ Design Decisions/ Educated Assumptions:**

* The spec did not specify what to do when two bullets interacted so I left them alone.
* I don’t think the spec mentioned where to place the player when he goes through a gate to a new sub level. In the demo program and mine, the player pops out where the gate was on the previous level.
* The spec was not clear on what to do if two goodies ended up being on the same square (kill a kleptobot carrying a goodie on a square with a goodie on it) so I left it alone

**Testing:**

**Player:**

I tested the player class by mostly just playing the game until the player did what I wanted when I pressed the appropriate keys. It took me a little while to get the player to stop going out of bounds. The isObstacle() function helped a lot with that and with all the other actor movements in the game.

I noticed player bugs when I would test other classes too and see if the player - actor interaction I was trying to fix was not working.

**Wall:**

Not much to say here. I ran the game and made sure the walls were in the correct place in the game compared to the demo and made sure objects couldn’t move through them. The second part I did mostly in the isObstacle() function though.

**Gate:**

I tested the gates by only loading the player, walls, and gates from all the sub levels to make sure the player was traversing the game correctly. Initially, I was getting a lot of bad access errors until I changed my data structure in StudentWorld from a vector of Actor pointers to a vector of vectors of Actor pointers.

**Jewel:**

I made sure the jewels were in the correct place, made sure they were granting the correct amount of points to the player upon being collected and that once they were collected, the exit appeared although this part relied mainly on testing the exit.

**Hostage:**

Nearly the same thing as the jewels except without making sure they granted the player points.

**Exit:**

I first commented out all the jewels and hostage to make sure the game could recognize that the exit would appear only once there were no jewels or hostages left. I did this to make sure the exit was first functioning (transporting the player to the next main level).

Once this worked, I added the jewels and hostages, and played the game to make sure the level only revealed the exit once I had collected the objectives.

**Bullet:**

I commented out everything except the player, bullet, and walls to make sure the bullet was firing in the correct place when I hit space and that it couldn’t go through walls.

Once I added everything back in, I made sure the bullet was doing the correct amount of damage to each actor, the bullet would damage the robot if a robot and an obstacle were in the same square, and that the all the right sounds were being played, etc.

**ExtraLifeGoodie:**

I added this and made sure it was in the correct place as the demo and made sure that it granted an extra life.

**RestoreHealthGoodie:**

Added this, made sure these were in the correct places, and that they granted the correct amount of health.

**AmmoGoodie:**

Added this, made sure these were in the correct places, and that they granted the correct amount of ammo.

**SnarlBot:**

The hardest part about implementing SnarlBots was getting their shooting patterns right. For a while, my SnarlBot would shoot even when the player was behind a wall. When I fixed that, it stopped shooting even when the player wasn’t behind a wall. I partially fixed it and it started shooting only when the player was right next to it. Finally, I fixed the clearShot() method and it was finally debugged and working.

The movement pattern for the SnarlBot was not too hard to implement since it follows a very predictable rhythm.

**KleptoBot:**

I commented everything out of the game except the player, walls, goodies, kleptobots, and the kleptobot factory. I then varied the 1 in 5 condition to make sure that it was actually making a difference. When that worked, I looked at its movement and compared it to the demo. It wasn’t the same of course, but I checked to see if it was at least the same amount of randomness(?).

At first, I was getting a bad access when the KleptoBot would pick up a goodie and encounter another. Then I realized it was because I did not put the condition that it could only pick up a goodie if it didn’t already have one.

The KleptoBots were not eating the goodies as frequently as they should have been at first either. Also, when I would kill the kleptoBot and it was carrying a goodie, the goodie would fly to its dead location since I used the moveTo method to facilitate it. I decided to change my template from merely setting the goodie invisible and making it reappear upon death to deleting the goodie and creating a new one when the Bot died.

**KleptoBotFactory:**

I made the game follow the same conditions as I did with the KleptoBot (see first paragraph for KleptoBot) and I varied the 1 in 50 condition.