**CS 32 Project 3** report.txt

Chang Liu

**Question 1** - Class Descriptions

*StudentWorld*

std::string formatDisplay(int score, int level, int lives, int health, int ammo, unsigned int bonus) { … };

* this function uses ostringstream to format the score to 7 digits and the level to 2 digits with 0’s filled in for the empty digits. For the other data, I just used the to\_string() function to convert the parameters passed in and appended everything at the end.
* I made this virtual since StudentWorld is the only class that needs to call it

void updateDisplay() { … }

* this function just sets the current level statistics to variables and passes those variables into the formatDisplay function
* the resulting string is then passed into the GameWorld’s setGameStatText function

void insert(Actor\* a) { … }

* this function is used specifically for the Factory class so that it can insert a KleptoBot / AngryKleptoBot into the StudentWorld’s private vector and have the StudentWorld track the inserted robots and call their respective doSomething() functions

virtual int init() { … }

* this function resets the data (number of Jewels, bonus and a boolean for if the level has been completed) at the start of a level. It then calls another function, loadLevel() which reads the grid of the data file and allocates Actors.
* this is virtual because the GameWorld that StudentWorld inherits from has the same init function but as a pure virtual one so we need to define it in the derived class

**virtual int move() { … }**

* this is virtual because of the same reasons for the init() function

update display

if Player is alive, doSomething

for every Actor

call their doSomething function if they’re alive

return if Player has died

return if level has been completed

removeDeadGameObjects()

reduce bonus

continue the game

virtual void cleanUp() { … }

* this is called upon by StudentWorld’s destructor; it iterates through StudentWorld’s vector of Actors and deletes every Actor object
* this is virtual because of the same reasons for the init() function

void removeDeadGameObjects() { … }

* similar to cleanUp, this iterates through every Actor but it only deletes those that are dead
* this is not virtual because the StudentWorld is the only class that needs the function since it is the manager of the Actors

void completed() { … }

int getJewels() { … }

* both these serve as helper functions for the Actor class; they are called upon by the Exit Actor to ensure that all the criteria for finishing a level are completed and then to notify the StudentWorld that it can advance to the next level

**Actor\* checkSpace(int x, int y, std::string search) { … }**

* this function is the core function used by all the classes to check if it can move to a space
* it returns whatever Actor, if any, that stands on the target square; the parameter search allows for prioritization of searching for a LivingActor or an Enemy, which cannot be occupied
  + this way, if there is a Goodie and a SnarlBot on one square, the function would prioritize the SnarlBot and therefore not let a Player move onto that square
* I made this non-virtual since no other class needs to call the function

for every Actor

if Actor’s coordinates are the same as the parameters

if the search string was “enemy”

dynamic cast to Enemy and return if not nullptr

if search was “goodie”

dynamic cast to Goodie and return if not nullptr

if search was “living”

dynamic cast to LivingActor and return if not nullptr

return the Actor

void createBullet(int x, int y, GraphObject::Direction dir) { … }

* spawns a Bullet class object in the square in front of the coordinates that were passed in and gives the Bullet to the StudentWorld vector to manage

Player\* getPlayer() { … }

std::vector<Actor\*> getActors() { … }

* these functions give access to the Actor pointer vector and the Player pointer held by the StudentWorld to other classes
* I made this non-virtual since no other class needs to call the function

bool canShoot(int x, int y, int dest, GraphObject::Direction dir, std::string check) { … };

* this function is used by SnarlBots and AngryKleptoBots to check if the Player is within their line of sight and not blocked by any obstructions
* the dest parameter is the coordinate of the Player (x coordinate if they are on the same row and y coordinate if same column) and the check parameter is passed in to make sure the Player is on the same line of sight as the robot
* for example, checking if a SnarlBot that is facing right can shoot would go like…

case right:

if check parameter is not horizontal, return false

if destination x coordinate is less than robot x return false (robot faces wrong way)

checkSpace for every square up until destination for hittable Actor

if so, return false

return true

int loadLevel();

* this function is responsible for adding all the Actors to the grid; it first formats the level string from GameWorld’s getLevel() function. Then it checks for a valid level data file from the formatted string. If so, it uses a nested loop to go through the 2D array and allocate Actors at each square as specified by the data file it reads from.

*Actor*

void convertDir(int& x, int&y, Direction dir) { … };

* prevents me from having to write a switch statement every time I need to move an Actor
* I would pass in the x and y coordinates of the Actor I want to move along with its direction and the function changes those coordinates to the Actor’s next step given the direction that it is facing
* I chose to make this function non-virtual and exclusive to the Actor class because all objects in the game inherit from Actor and this function would not be different for certain actors since all are on the same coordinate grid

virtual void doSomething() = 0;

* I made the doSomething() function pure virtual in my Actor class because it should never be called on an Actor class and only on classes that inherit from Actor

virtual bool canOccupy() { … };

* helps check if a moving Actor could move to a certain block, by checking this function
* I set it to return false by default in the Actor class because the majority of Actors (KleptoBot, AngryKleptoBot, Player) cannot be on the same square as another moving actor
* I made this virtual to handle those exceptions of Actors that can be occupied *e.g.* Jewel, Health Goodie, Life Goodie, Ammo Goodie

virtual bool hittable() { … };

* serves to let the Bullet class know which objects it can hit and die on, returning true for those that can be hit
* I made it virtual and true by default because most of the Actors can be hit by bullets; I would just add a redefinition in the classes that are not hittable (*e.g.* Goodies, Jewel)

virtual bool isHit(int damage) { … };

* triggers whenever a Bullet hits a hittable object (described by the previous function)
* I made it virtual because all Actors behave differently when hit, *e.g.* they may lose health
* I also chose to not make this pure virtual so that Actors that are hittable by default do nothing when they’re hit, *e.g.* a Wall, Factory does nothing when hit but it can still be hit

StudentWorld\* getWorld() const { … };

* this function returns a pointer to the StudentWorld object so that each Actor can access its public member functions
* I did not make it virtual because there is only one StudentWorld object so all Actors should be accessing the same world

void setDead() { … };

* sets the Actor’s private boolean variable m\_isAlive to false, indicating death
* I did not make it virtual because all Actors die in the same way by being removed from the grid; I handled individual behavior like dropping an item in the isHit() functions

bool isAlive() { … };

* this function checks to see if the Actor is still alive and is called upon by StudentWorld so that dead Actors can be removed
* I did not make it virtual because the check should remain the same for all classes, by only checking the private member m\_isAlive

*Actor → Wall*

virtual void doSomething() { };

* I made the body of this inherited function empty because the spec stated that a Wall should do nothing during each tick and because the abstract base class does not have it defined

*Actor → Hole*

virtual void doSomething() { };

* same reasons as *Wall*

*Actor → Bullet*

**virtual void doSomething() { … };**

check if Bullet is dead

search current space for LivingActor

call the LivingActor’s isHit function, die, and return

search current space for an Actor

if Actor is hittable, call the isHit function, die, and return

move

virtual bool canOccupy() { … };

* I redefined the base class’ canOccupy() function so that it returns true for Bullets; this way, other Actors can be on the same square as a Bullet so the Bullet can call their isHit function

virtual hittable() { … };

* I redefined this to return false for Bullets which means that Bullets traveling towards each other will pass one another and continue in their respective directions

*Actor* → *Exit*

virtual void doSomething() { … };

* checks if it isn’t visible to begin with and if the amount of Jewels the Player acquired equals the amount of Jewels originally on the level; if so, set itself to visible
* if the Player is occupying the same square as the Exit and it is visible, tell the StudentWorld the level is completed

virtual bool canOccupy() { … };

* this returns true because other Actors can occupy the same square as the Exit; this is needed for the Player especially

virtual hittable() { … };

* this redefines the base class function and returns false because Bullets should pass through the Exit

*Actor* → *Jewel*

virtual void doSomething() { … };

* first checks if it’s dead in which case it will return immediately
* if Player is on the same coordinates as the Jewel, set itself to dead and tell the StudentWorld to increase the score and the Player to increase his jewel count

virtual bool canOccupy() { … };

* returns true because other Actors can occupy the same space as the Jewel

virtual hittable() { … };

* returns false because Bullets should go through a Jewel

*Actor → Factory*

bool countRegion() { … };

* checks its private member vector of Actors (which holds pointers to robots spawned from the specific Factory) and increments a count variable for every robot that’s within the specified distance from Factory
* returns true if the count is less than 3 robots
* this is the only class that needs to count robots so it is not a virtual function

virtual void doSomething() { … };

* if the countRegion() function returns true, it uses cstdlib’s rand() function to have a 1/50 chance of spawning a new KleptoBot (angry or not depending on the Factory’s m\_angry boolean) which will be added to the StudentWorld vector and the Factory’s own vector

*Actor → Goodie*

virtual void doSomething() = 0;

* I made this pure virtual because the Goodie class’ doSomething() function should never be explicitly called, only on derived classes

virtual bool canOccupy() { … };

* this returns true because other Actors can occupy the same square as the Goodie and all its derived classes (Health, Ammo, Life)

virtual hittable() { … };

* returns false because Bullets should pass through Goodies

*Actor → Goodie → Health*

*Actor → Goodie → Life*

*Actor → Goodie → Ammo*

virtual void doSomething() { … }

* for all the Goodie class’ derived classes, the doSomething function first checks if it is dead in which case it would return immediately
* then it checks if the Player is on the same square and if the Goodie itself is visible in which case the StudentWorld would be notified to give the player the particular perks and the Goodie would set itself to dead

*Actor → LivingActor*

virtual void doSomething() = 0;

* I made this pure virtual because the LivingActor class’ doSomething() function should never be explicitly called, only on derived classes

int getHealth();

int changeHealth(int amt);

* these two functions serve as helper functions for derived classes and other classes (*e.g.* Health Goodie) to access the LivingActor’s hit points
* I made it non-virtual because getting / changing the health of a LivingActor should not differ in the derived classes

virtual void isHit(int damage);

* here I redefined the Actor class’ isHit function so that the LivingActor actually takes damage when hit by a Bullet
* since isHit handles the death of LivingActors, I made it virtual again so that derived classes can die in different ways *e.g.* a KleptoBot dropping its item upon death

*Actor → LivingActor → Boulder*

virtual void doSomething() { };

* I made the body of this inherited function empty because the spec stated that a Boulder should do nothing during each tick and because the abstract base class does not have it defined

bool push(Direction dir);

* first checks the next square in the direction that the Player is pushing for an Actor; only if the check function returns a nullptr will the Boulder move; or else if the space could be dynamically cast to a Hole, the Boulder will move there and both Actors will be set to dead
* this function is not virtual because it is exclusive to the Boulder class

*Actor → LivingActor → Player*

virtual void doSomething() { … };

* this function handles user input to control the Player. It first calls the canMove function when a directional arrow key is pressed to check if the Player can move to that spot. If it returns true, then the Player will move. Similarly, pressing SPACE will first check if a Player has Ammo and if so, then he will shoot a Bullet. Lastly, the ESC button will just set the Player to dead.

int getAmmo();

void reload(int amt);

* both these functions are for interacting with the Ammo Goodie so that picking it up will update the Player’s private ammo data
* these are not virtual because the Player is the only class to have a finite Ammo count

int getJewels() { … }

void incJewels() { … }

* both these functions are for interacting with the Jewel Goodie so that picking it up will update the Player’s private jewel data
* these are not virtual because the Player is the only class that can collect Jewels

virtual void isHit(int damage) { … };

* I redefined the base class function to implement the Player specific behavior like playing the sounds
* it checks if the m\_health data member of Player inherited from LivingActor has reached 0 or below in which case it will set the Player to dead

void shoot() { … };

* this calls a StudentWorld function to create a Bullet in front of the Player

**bool canMove(int& x, int& y, Direction dir) { … };**

convert coordinates of Player to get the next space according to where he faces

check that space for an Enemy class

return false

check the space for any Actor

if Actor is occupiable

return true

check space for Boulder

if Boulder’s push function returns true

return true

return false (meaning there is an unoccupiable Actor there)

* this function is not virtual because I only need it for my Player class; the Enemy classes handle the move checking in their doSomething() function. I did this to save code since the Player’s doSomething() uses a switch statement and I would then have to repeat this code 4 times for the possible directions.

*Actor → LivingActor → Enemy*

int getRest() const { … }

int getTicks() const { … }

void setTicks() { … }

* these three functions are to manage the internal ticks of each Enemy; rest is the amount of ticks an Enemy must wait before being allowed to move
* since they run on the same tick system, I did not make these functions virtual
* setTicks checks if ticks the Enemy has waited is equal to its required rest; if so, it will reset the ticks waited to 0 and if not it will increment the ticks waited

void isHit(int damage) = 0;

* I made this a pure virtual function since it should never be called on an object that’s just of class Enemy. Each derived class of Enemy also behaves differently for this function since it also manages the death behavior.

virtual void doSomething() = 0;

*Actor → LivingActor → Enemy → SnarlBot*

**virtual void doSomething() { … };**

if dead, return

get coordinates and direction

if ticks waited < ticks supposed to wait

increment ticks

else

get Player coordinates

if Player is in the same column

use StudentWorld to check if SnarlBot can shoot

shoot if true and return

if Player is in the same row

use StudentWorld to check if SnarlBot can shoot

shoot if true and return

check space in front

if can be occupied, move there

else turn 180 degrees

virtual void isHit(int damage) { … };

* this will play the appropriate sounds for a SnarlBot when its hit and when it dies
* it checks if the m\_health data member of SnarlBot inherited from LivingActor has reached 0 or below in which case it will set the SnarlBot to dead

void shoot() { … };

* tells the StudentWorld to create a Bullet in front of the SnarlBot and plays the appropriate sound

*Actor → LivingActor → Enemy → KleptoBot*

virtual void isHit(int damage) { … };

* this will play the appropriate sounds for a KleptoBot when its hit and when it dies
* it checks if the m\_health data member of KleptoBot inherited from KleptoBot has reached 0 or below in which case it will set the SnarlBot to dead

**virtual void doSomething() { … };**

if dead, return

get coordinates and direction

if supposed to rest

rest and return

if is AngryKlepto

if Player is in the same column

use StudentWorld to check if SnarlBot can shoot

shoot if true and return

if Player is in the same row

use StudentWorld to check if SnarlBot can shoot

shoot if true and return

check current square for visible Goodie if it’s not holding an item already

if so, 1/10 chance function

get the item and make it invisible

if it hasn’t moved the distance before turning yet and hasn’t been blocked

check space for LivingActor

indicate it’s been blocked

check space for Actor

if none, move there and update the distance moved

else indicate it’s been blocked

if it has moved the distance or has been blocked

set blocked to false

reset distance before turning

loop, get a random direction and map that to a bool

if direction has already been mapped, find another direction

check space in front of robot for LivingActor

check space in front for Actor

move if none

if 4 directions have been mapped set direction to the first one

*Actor → LivingActor → Enemy → KleptoBot → AngryKleptoBot*

virtual void isHit(int damage) { … };

* I redefined KleptoBot’s isHit function because AngryKleptoBot must drop an item upon death if it was holding one in addition to KleptoBot’s other behavior

**Question 2** - Bugs

1. The game runs on Xcode fine but it encounters bugs because of the vector implementation in Visual C++
2. Sometimes, the SnarlBot can move onto the same square as a KleptoBot. Other times it would change directions as expected; I think this is an issue of timing.
3. In the second to last commit I had, the KleptoBots would sometimes be able to move diagonally. In the most recent commit, I have not seen this behavior yet so I believe I may have fixed it.

**Question 3** - Design Decisions

The following were features that were not specified in the spec but ones that I decided to implement because of the sample version’s behavior

1. KleptoBots will appear behind the Exit if they are on the same square and the Exit is visible
2. Two Goodies can be on the same space
3. KleptoBots and SnarlBots can stand on the same square as a Jewel
4. SnarlBots can occupy the same square as a Goodie
5. The ammo count resets on each level

**Question 4** - Testing

*Actor*

* To test this base class, I made sure the Bullets disappeared upon hitting any derived class of Actor that had not redefined the hittable() function.

*- Wall*

* I moved my player against the wall to test if the canOccupy() function inherited from the base class worked and blocked the Player. I also shot Bullets at the Walls to make sure the Bullets died upon impact and nothing happened to the Walls.

*- Hole*

* I made sure Bullets would pass through a Hole and that robots and the Player could not step on one. I also made sure that both a Hole and Boulder would die when occupying the same square and thus make that square occupiable.

- *Exit*

* I moved the Player to the Exit when it was invisible and the Jewels were not collected to ensure it didn’t trigger when the level wasn’t completed. I then tested to make sure it appeared the moment the last Jewel was collected and that the sound played only once. Finally, I shot at it to make sure the Bullet would pass through.

*- Jewel*

* I made sure the Player and all the robots could occupy the same square as the Jewel but only the Player could pick it up and make it disappear. Also made sure Bullets would pass through it.

*- Factory*

* I manually counted the robots in the region to make sure the Factory wouldn’t spawn any new ones if there were at least 3 present or if the current space was occupied by a newly spawned robot. I also fired at the Factory to make sure Bullets would die and do nothing upon impact (unless there was a KleptoBot on the Factory). Last I made sure no other Actors could occupy the Factory square, except for a newly born KleptoBot, and that the Factory spawned the right robots.

*- Goodie*

* I made sure these could be picked up by a KleptoBot (I turned off the randomness of a bot picking it for faster testing) and again after a KleptoBot had dropped it. I also had to make sure they couldn’t be picked up when they were already “stolen” and that they were dropped at the exact position where the KleptoBot died.

*- - Health*

* I ensured that this would update the Player’s health to 20 and nothing more / less when picked up.

*- - Ammo*

* I ensured that this updated the Ammo display.

*- - Life*

* I ensured that this updated the Lives display and I pressed the ESC key to die on purpose to ensure I could die one more time than the default.

*- LivingActor*

*- - Player*

* I made sure the Player could pick up all the Goodies that were visible. I also made sure the game would end immediately when the Health reached 0%. I checked the Boulder specs to make sure the Player could only push the Boulders in the specified situations.

*- - Enemy*

*- - - SnarlBot*

* I moved the Player to coordinates that were not in the SnarlBot’s line of sight to make sure it wouldn’t fire. I made sure the SnarlBot would prioritize shooting the Player among all other actions and that it didn’t move on every tick by changing the game speed to see. Last, I ensured the SnarlBot turned 180 degrees upon reaching an obstruction.

*- - - KleptoBot*

* I made sure the KleptoBot was actually being klepto by removing the randomness of it picking up an item for accurate testing purposes. I made sure the item could be picked up after it died and that it would not pick up more than one item. I also slowed down the game speed and made sure the KleptoBot would not move during a tick only if it was surrounded on all sides.

*- - - AngryKleptoBot*

* I used the same test cases for KleptoBot to test the AngryKleptoBot. I also made sure it prioritized shooting the Player if he was within its line of sight above all other decisions.