Jason Tieu

304047667

Project 3 Report

*1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Bug Blast are able to sneeze, and each type of actor sneezes in a different way.”*

**StudentWorld**

StudentWorld() – This is the constructor for my StudentWorld class. I primarily created this to push the level files into a vector so they can be loaded when I call my LoadLevel() function.

~StudentWorld() – This is the destructor for my StudentWorld class. This is used to delete the member data such as vectors, lists, and pointers.

virtual int init() – This is a virtual function from GameWorld. It is a premade function that is called to set up a level.

virtual int move() – This is a virtual function from GameWorld. It is called every tick, and is used to call the functions from each Actor that determines whether an Actor is doing an action or not.

...

repeatedly:

if actor is not dead,

call actor's performAction()

if actor is Exit,

set exit pointer to actor

if player is dead,

return player died

if level is completed,

increase score

play sound for level finished

return level finished

if player is not dead

call player's performAction()

remove dead game objects

decrement level bonus

if all zumi are dead,

reveal maze exit

play reveal exit sound

if player is dead

return player died

if level is completed

increase score

player sound for level finished

return level finished

return continue game

virtual void cleanUp() – This is a virtual function from GameWorld. It is called when the Player is killed for any reason. cleanUp gets rid of all Actors in the current level and ‘resets’ the level.

int LoadLevel() – This function is used to load the level data for whichever the current level is. It creates the Actors required for the level and saves the numerical options needed.

...

if current level is the first level and did not load

return no first level

if current level not first level and cannot load

return player won

if cannot load level

return error

if load level successful

repeatedly:

repeatedly:

if level item is player

create new Player

if level item is perma\_brick

push new PermanentBrick onto vector

if level item is destroyable\_brick

push new DestroyableBrick onto vector

if level item is exit

push new Exit onto vector

if level item is simple\_zumi

push new SimpleZumi onto vector

if level item is complex\_zumi

push new ComplexZumi onto vector

...

return continue game

void UpdateDisplayText() – This function is used to update the score, lives left, bonus, etc for the User to see when playing the game.

string ScoreBoard(int Score, intLvl, int Lives, int bonus) – this function displays the score in a specific layout.

bool PlayerIsNotDead() const – this function returns if the player is dead or not.

bool ActorIsNotDead(Actor& a) const – this function returns if a particular Actor is dead or not.

bool isCurrentLevelCompleted() – this function determines if the current level has been completed or not.

void IncreaseScoreByBonus() – this function increases the score by the bonus.

void removeDeadGameObjects() – this function gets rid of any objects that are dead

bool allZumiDead() – this function checks if all Zumis currently in the level are dead or not.

void revealExitinMaze() – this function reveals the exit of the maze

int getID\_of\_other\_in\_square(int x, int y) – this function gets the ID of an Actor in the particular x,y coordinates, if there is not Actor then it returns a distinct value

bool occupy\_with\_player(int x, int y) – this function determines whether or not the x,y coordinate is occupied by the Player

void insert\_Actor(Actor\* a) – this function inserts an Actor into the member vector stored by the StudentWorld class. This function is primarily used outside of the StudentWorld class.

Player\* get\_Player() – this function retrieves the player pointer for easier access outside the StudentWorld class

Actor\* get\_Actor(int id, int x, int y) – this function gets a particular Actor if it fulfills the ID, x and y arguments

int get\_extra\_lifetime() – this function gets the value stored by m\_extra\_lifetime

int get\_walk\_thru\_lifetime() – this function gets the value stored by m\_walk\_thru\_lifetime

int get\_walk\_thru\_ticks() – this function gets the value stored by m\_walk\_thru\_ticks

int get\_inc\_simul\_lifetime() – this function gets the value stored by m\_inc\_simul\_lifetime

int get\_inc\_simul\_ticks() – this function gets the value stored by m\_inc\_simul\_ticks

int get\_num\_sprayers() – this function gets the value stored by m\_num\_sprayeres

int get\_simple\_zumi\_ticks() – this function gets the value stored by m\_simple\_zumi\_ticks

int get\_complex\_zumi\_ticks() – this function gets the value stored by m\_complex\_zumi\_ticks

int get\_complex\_zumi\_search\_dist() – this function gets the value stored by m\_complex\_zumi\_search\_dist

int get\_prob\_goodie() – this function gets the value stored by m\_prob\_goodie\_overall

int get\_prob\_extra\_life() – this function gets the value stored by m\_prob\_goodie\_extra\_life

int get\_prob\_walk\_thru() – this function gets the value stored by m\_prob\_goodie\_walk\_thru

int get\_prob\_simul\_bugspray() – this function gets the value stored by m\_prob\_goodie\_simul\_bugspray

**Actor (derived from GraphObject)**

Actor(intimageID, intstartX, intstartY, StudentWorld\* s) – this is my constructor for my Actor class. It is called by it’s derived classes.

void set\_status(bool s) – this function sets the status (alive(true)/dead(false)) for an Actor

bool get\_status() const – this function returns the status of an Actor

virtual int performAction() = 0; I made this function pure virtual since the Actor should be an abstract base class (ABC). This function will be constantly called by move() in StudentWorld, so each Actor should have their own separately defined version of performAction().

StudentWorld\* get\_SWorld() const – this function returns a pointer to StudentWorld so Actors can access functions of StudentWorld

int get\_ActorID() const – this function returns the ID of an Actor

void set\_Actor\_vis(bool t) – this function sets the Actor visibility. Also sets a private data called m\_vis to what the actor visibility is since we are unable to get the Actor visibility from GameWorld.

bool get\_Actor\_vis() const – this function returns m\_vis

**Player (derived from Actor)**

Player(intstartX, intstartY, StudentWorld\* s) – this is the constructor for my Player class. It passes the arguments to the constructor for the Actor class.

\*\*virtual int performAction() – this is a distinctly defined version of performAction() specifically for the Player. It is called via the Actor class, but since it is virtual it will be able to pick out Player’s performAction() function

if player is dead

return player died

...

if can walk through bricks

decrement walk through ticks

if can set extra bugsprayers

decrement extra bugsprayers ticks

...

if key is pressed

switch key

case down:

if able to move to square

move down

break

case up:

if able to move to square

move up

break

case left:

if able to move to square

move left

break

case right:

if able to move to square

move right

break

case space:

if can place BugSprayer

create new BugSprayer

decrement available BugSprayer amount

break

return continue game

void set\_max\_bug\_sprayers(int t) – sets the max bug sprayer amount to the int being taken in

void get\_max\_bug\_sprayers() – this function returns the max bug sprayers amount

void set\_avail\_bug\_sprayers(int t) – sets the available bug sprayer samount to the int being taken in

void get\_avail\_bug\_sprayers() – returns the available bug sprayers amount

void set\_walk\_thru\_bricks(bool check) – this function sets the status for the walk through bricks power

bool get\_walk\_thru\_bricks\_status() const – this function returns the status of the walk through bricks power

void set\_extra\_bug\_sprayers(bool check) – this function sets the status for the extra bug sprayers power

bool get\_extra\_bug\_sprayers() const – this function returns the status of the extra bug sprayers power

void set\_walk\_thru\_ticks(int check) – this function sets the amount of time the walk through bricks power is activated

void set\_extra\_bug\_sprayers\_ticks(int check) – this function sets the amount of time the extra bug sprayers power is activated

**DestroyableBrick (derived from Actor)**

DestroyableBrick(int startX, int startY, StudentWorld\* s) - constructor for the Destroyable Brick class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's perform action is pure virtual. This is distinctly defined for DestroyableBrick. It is called once per tick by StudentWorld's move() function.

-------------------------------------------------------------------------------------------------------------------------------

**PermanentBrick (derived from Actor)**

PermanentBrick(int startX, int startY, StudentWorld\* s) - constructor for the Permanent Brick class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's perform action is pure virtual. This is distinctly defined for PermanentBrick. It is called once per tick by StudentWorld's move() function. In this case, PermanentBrick's performAction() immediately returns since it has no role besides from being a brick.

-------------------------------------------------------------------------------------------------------------------------------

**Exit (derived from Actor)**

Exit(int startX, int startY, StudentWorld\* s) - constructor for the Exit class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's perform action is pure virtual. This is distinctly defined for Exit. It is called once per tick by StudentWorld's move() function.

bool get\_exit\_sound() - returns a bool status of whether or not the exit sound has been played or not

void set\_exit\_sound(bool t) - sets the status of the exit sound to the argument t.

-------------------------------------------------------------------------------------------------------------------------------

**BugSprayer (derived from Actor)**

BugSprayer(int startX, int startY, StudentWorld\* s) - constructor for the BugSprayer class, passes its arguments into the Actor constructor

\*\*virutal int performAction() - virtual since Actor's performAction() is pure virtual. BugSprayer's performAction() has its own actions and is called once per tick by StudentWorld's move() function.

if BugSprayer is alive

if lifetime > 0

decrement lifetime

if lifetime equal to 0

play spray sound

create BugSpray at this location

...

repeatedly

if square is valid to the right

create new BugSpray in square

repeatedly

if square is valid to the left

create new BugSpray in square

repeatedly

if square is valid up

create new BugSpray in square

repeatedly

if square is valid down

create nwe BugSpray in square

set status to dead

increment Player's available bugsprayers

return continue game

void set\_sprayer\_lifetime(int t) - sets the lifetime for the bugsprayer to the value passed in

int get\_sprayer\_lifetime() returns the lifetime of the sprayer

-------------------------------------------------------------------------------------------------------------------------------

**BugSpray (derived from Actor)**

BugSpray(int startX, int startY, StudentWorld\* s) - constructor for the BugSpray class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's performAction() is pure virtual. BugSpray's performAction() has its own actions and is called once per tick by StudentWorld's move() function.

...

if alive

if lifetime > 0

decrement lifetime

...

if occupy with DestroyableBrick

set DestroyableBrick to dead

if occupy with BugSprayer

set BugSprayer lifetime to zero

if occupy with player

set Player to dead

if occupy with SimpleZumi or ComplexZumi

set SimpleZumi to dead

play enemy die sound

increase score

determine if goodie is dropped

set status to dead

return continue game

void determine\_Goodie\_type() - a function used to help determine which goodie (three types) will be dropped by generating a random number and comparing it to the values given by StudentWorld

...

sort goodie vector in ascending order

repeatedly:

if goodie value is zero

set bool check\_for\_zero to true

break

...

if goodie will spawn

...

if check\_for zero is true

if chance of spawn < middle goodie

spawn goodie (middle)

if chance of spawn > middle goodie

spawn goodie (largest)

else

if chance < least

spawn goodie (least)

if chance > least and chance < big

spawn goodie (mid)

if chance > big

spawn goodie (largest)

void spawn\_goodie(int x) - this function spawns a particular goodie that matches the argument x.

-------------------------------------------------------------------------------------------------------------------------------

**ExtraLifeGoodie (derived from Actor)**

ExtraLifeGoodie(int startX, int startY, StudentWorld\* s) - constructor for the ExtraLifeGoodie class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's performAction() is pure virtual. ExtraLifeGoodie's performAction() has its own actions and is called once per tick by StudentWorld's move() function.

-------------------------------------------------------------------------------------------------------------------------------

**WalkThruWallsGoodie (derived from Actor)**

WalkThruWallsGoodie(int startX, int startY, StudentWorld\* s) - constructor for the WalkThruWallsGoodie class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's performAction() is pure virtual. WalkThruWallsGoodie's performAction() has its own actions and is called once per tick by StudentWorld's move() function.

-------------------------------------------------------------------------------------------------------------------------------

**IncSimulSprayerGoodie (derived from Actor)**

IncSimulSprayerGoodie(int startX, int startY, StudentWorld\* s) - constructor for the IncSimulSprayerGoodie class, passes its arguments into the Actor constructor

virtual int performAction() - virtual since Actor's performAction() is pure virtual. IncSimulSprayerGoodie's performAction() has its own actions and is called once per tick by StudentWorld's move() function.

-------------------------------------------------------------------------------------------------------------------------------

**SimpleZumi (derived from Actor)**

SimpleZumi(int startX, int startY, StudentWorld\* s) - constructor for the SimpleZumi class, passes its arguments into the Actor constructor

\*\*virtual int performAction() - virtual since Actor's performAction() is pure virtual. SimpleZumi's performAction() has its own actiosn and is called once per tick by StudentWorld's move() function.

if alive

if occupy square with Player

set Player status to dead

play dead sound

return player died

if move ticks > 0

decrement move ticks

else

switch (current direction)

case up:

if square is valid

move up;

randomize current direction

break

case down:

if square is valid

move down;

randomize current direction

break

case left:

if square is valid

move left;

randomize current direction

break

case right:

if square is valid

move right;

randomize current direction

break

default:

randomize current direction

set move ticks to default

return continue game

-------------------------------------------------------------------------------------------------------------------------------

**ComplexZumi (derived from Actor)**

ComplexZumi(int startX, int startY, StudentWorld\* s) - constructor for the ComplexZumi class, passes its arguments into the Actor constructor

\*\*virtual int performAction() - virtual since Actor's performAction() is pure virtual. ComplexZumi's performAction() has its own actiosn and is called once per tick by StudentWorld's move() function.

if alive

if occupy square with Player

set Player status to dead

play dead sound

return player died

if move ticks > 0

decrement move ticks

else

check horizontal and vertical distance from player if within search distance

set smell to truee

if smell is true

do breadth-first search

if path exits

move to first x and y in path

if not within search distance or path doesn't exist

switch (current direction)

case up:

if square is valid

move up;

randomize current direction

break

case down:

if square is valid

move down;

randomize current direction

break

case left:

if square is valid

move left;

randomize current direction

break

case right:

if square is valid

move right;

randomize current direction

break

default:

randomize current direction

set move ticks to default

return continue game

-------------------------------------------------------------------------------------------------------------------------------

*2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g., “I wasn’t able to implement the Exit class.” or “My Complex Zumi doesn’t work correctly yet so I just treat it like a Simple Zumi right now.”*

-There is a bug with picking up the Simultaneous Bug Sprayer goodie. There are times where the Player is able to place more or less than the max amount of goodies allowed. After extensive play testing, I was not fully able to pinpoint the main reason. I decided that there wasn't much I could do to fix it with the time remaining to submit the game.

- I was unable to get the Complex Zumi's breath-first search to work properly. It is able to follow the player, but the game keeps crashing and there seems to be issues with how the Complex Zumi is choosing to move or not. It is moving onto squares that contains Bricks and Bugsprayers even though I tried to restrict it. Due to these issues, I decided to make it act as a Simple Zumi right now, but I am basing its movement ticks on the option for Complex Zumi given by the Level data.

-------------------------------------------------------------------------------------------------------------------------------

3. A list of other design decisions and assumptions you made, e.g., “It was unspecified what to do in situation X, and this is what I decided to do.”

- I made a LoadLevel() function in the StudentWorld class. With this, I also decided to make a vector that contains the levels (i.e. level00.dat, level01.dat...). These were pushed\_back onto the vector in the StudentWorld constructor. After each level, the next element in the vector is accessed. I decided to do this as a way to navigate through the levels. My only main concern with this is that in extra levels must be added via the constructor since anywhere else could cause copies of the level to be played again due to navigating through the vector.

- When testing the Bug Blast game provided to us, I was never able to beat all the levels. So I decided to exit the program when the last level is completed.

- I decided not the follow the specification document exactly for many cases. One would be how during the Player's performAction(), it should check if he's occupying the same location as a Zumi or Destroyable Brick (without the goodie power on) to return that he is dead. While doing so, I kept getting crashes when doing it this way. Instead, I decided to divert the tasks to the respective actors, so each Destroyable brick and Zumi tests to see if it is occupying the same location as the player. Although this may be a bit redundant, the computations do not seem too expensive since there was no notable lag when deciding the implement the task this way.

-------------------------------------------------------------------------------------------------------------------------------

*4. A description of how you tested each of your classes (1-2 paragraphs per class)*

**StudentWorld**

I tested out this class by continually adding pieces of code and playing the game. I constantly had to check if the init() function was generating the required Actors for the level. I did this using cout statements to give me an understanding if a certain piece of code is reached or not during testing. For the move() function, I checked if each part of code is being reached out not using cout statements as well. I made sure to check every instance, such as when all Zumis die, the exit should appear. Also I made sure everything was synced properly, as in all the Actors are receiving the same information from StudentWorld. I ensured this by using helper functions that allowed me to access specific actors and output their status and information. I also accessed specific actors in other actor classes to make sure the information was the same for StudentWorld and the actors. Also, I made sure the StudentWorld was deleting game objects that were no longer alive properly. I did this by making basic destructors for my Actors that simply outputted something like "~(Actor name)". Although it is not in my final build, during testing it was incredibly helpful to see if the actor was deleted once it was deemed "dead".

**Actor**

Since my Actor class is an abstract base class, I could not have tested it directly. I had to use it's derived classes to make sure it was working properly. Since I implemented this class with set and get functions, I continuously made calls to these in the derived classes to make sure the Actor class was always working properly. Since the Actor's constructor was always called by the derived classes, I made sure the ID and (x,y) were always corresponding with the layout from the levels. I used mainly cout statements to ensure the Actor class was not doing anything strange that would negatively affect it's derived classes.

**Player**

The player was tested out the most. To ensure it was working exactly as it was supposed to, I tested out how the Player reacts to every other Actor. For example, I would test the Player only with destroyable bricks, to make sure the Player could not walk through them without the goodie. Once that was correct, I made sure it could walk through them while with a goodie. Then I would wait until the goodie power expired and see if I died while remaining in the destroyed brick. I did these kinds of tests with every actor. Once I was for sure that the player interacted properly with other actors, that was when I started fully reconstructing the level and seeing how the player interacted with the other actors while multiple types of actors were present.

**DestroyableBrick**

The DestroyableBrick only required simple testing. I made sure it interacted with the player properly. For my implementation, the DestroyableBrick is the one that checks if the player is on its square. I made sure this was working properly, and that the player's status was set to DEAD if it was on the same square as an alive destroyable brick while not having the walk through brick goodie. I also made sure the DestroyableBrick could not be passed through by SimpleZumi and ComplexZumi by using conditional statements in the Zumi classes. Also, I made sure the DestroyableBrick worked with bugspray, since it had to be destroyed if it was on the same square as a bugspray. This is handled by the bugspray.

**PermanentBrick**

The PermanentBrick class required very little testing. I only had to make sure that the player, simple and complex zumis can never go through it. This simply required conditional statements in the respective classes. Other than that, the permanent brick did not require any further testing.

**Exit**

For the Exit class, I made sure it was being used properly by the StudentWorld class. In the move() function, I made an Exit pointer that dynamically casted the correct Actor pointer. This allowed me to easily access the Exit and made sure it would be visible/invisible at the appropriate times. I made sure when the player stepped on a visible Exit, then the level would be completed. It was important to make sure the StudentWorld received this information correctly to prevent any issues with playing the required sounds needed.

**BugSprayer**

For the BugSprayer, I had to make sure it was able to properly be created when the Player pressed SPACE. Once that worked, I had to make sure the BugSprayer's lifetime decremented each tick until it reached zero. Once the lifetime ran out, it had to generate BugSpray in up to 2 squares in each direction (UP/DOWN/LEFT/RIGHT). To ensure the correct amount of BugSpray was created, I made several conditional statements to ensure that the BugSprayer would only generate BugSpray within the level's bounds and the square isn't a permanent brick. After it generated the BugSpray properly, the BugSprayer had to be destroyed. I made sure this was working correctly by making a temporary destructor for the BugSprayer outputting a statement signifying that it was destroyed.

**BugSpray**

For the BugSpray, I had to make sure it was interacting with specific Actors properly. The main actors would be the Player, SimpleZumi, ComplexZumi, DestroyableBrick, and BugSprayer. When in contact with a Player, I had to make sure that meant the player would die. I tested this by constantly making the Player run into BugSpray and making sure he died. For SimpleZumi and ComplexZumi, I made sure they also died when in contact with BugSpray. I did this by making the Zumi's idle and hitting them with BugSpray and making sure they died every time. DestroyableBrick also had to be destroyed when in contact with a BugSpray. I tested it the same as for the Zumi classes. As for BugSprayer, I had to replicate a 'chain reaction' effect where a BugSprayer would immediately detonate when in contact with BugSpray. I decided to do this by making a BugSprayer pointer that dynamically casted the correct Actor pointer if BugSpray was occupying the same square as that Actor. Then I set the BugSprayer's lifetime to 0, thus making the BugSprayer generate BugSprayer. This worked out perfectly. I made sure the chain reaction had no issues by giving the Player unlimited BugSprayers and constantly placing BugSprayers within proximity of each other.

As for dropping Goodies, I created functions to check the chance for a Goodie to drop when a Zumi was killed. I ensured this worked correctly simply be modifying the values in the level data and seeing if my expectations of the drop matched what actually happened. I made sure the goodie dropped on the square that the Zumi was killed.

**ExtraLifeGoodie**

For the ExtraLifeGoodie class, all that was required for testing was to see if when it spawned, it would be destroyed once it's timer was up. If the timer was up, I made sure that the player could not interact with the expired goodie. While the goodie was still available, I made sure the Player gained accessed to its functions if it was on the same square as the goodie. This meant increasing the player's life by one, increasing player's score by 1000, and playing a sound that signified that the goodie was accepted by the player.

**WalkThruWallsGoodie**

For the WalkThruWallsGoodie class, all that was required for testing was to see if when it spawned, it would be destroyed once it's timer was up. If the timer was up, I made sure that the player could not interact with the expired goodie. While the goodie was still available, I made sure the Player gained accessed to its functions if it was on the same square as the goodie. This meant that the player was able to walk through DestroyableBricks while the goodie was activated (once the player got the goodie, a timer would go down, signifying how long the player has to walk through bricks), increasing player's score by 1000, and playing a sound that signified that the goodie was accepted by the player.

**IncSimulSprayerGoodie**

For the IncSimulSprayerGoodie class, all that was required for testing was to see if when it spawned, it would be destroyed once it's timer was up. If the timer was up, I made sure that the player could not interact with the expired goodie. While the goodie was still available, I made sure the Player gained accessed to its functions if it was on the same square as the goodie. This meant that the player was able to place down more BugSprayers while the goodie was activated (once the player got the goodie, a timer would go down, signifying how long the player has access to more BugSprayers), increasing player's score by 1000, and playing a sound that signified that the goodie was accepted by the player.

**SimpleZumi**

The SimpleZumi was required to be able to move in random directions. After getting the required information from StudentWorld to determine how often it could move, I used a switch statement to determine its movement. The SimpleZumi starts out with a random current direction variable (1 - up, 2 - down, 3 - left, 4 -right). For each case in the switch statement, I made sure that the SimpleZumi could only move when the square it was going to move to did not contain any type of Brick or a BugSprayer. This was done using conditional statements. The current direciton variable will randomize once again after each possible move, giving the illusion that that SimpleZumi is randomly moving whenever it is allowed to move. I also made sure that when it was in the same square as a player, then the player's status would be dead and would immediately return that the player died.

**ComplexZumi**

The ComplexZumi was required to be able to move in random directions. I had to check if it's distance from the player was within a "smell range". The smell range was determined by the level's data. If so, then I had to conduct a breadth-first search to determine the most efficient route to the player's location. I was unable to get the search to properly work, so I decided to make the ComplexZumi act as a SimpleZumi, while how often it could move was determined by the ComplexZumi movement ticks given by the level.