Simran Dhaliwal

CS 32 Nachenberg

Project 3 Report

1. Here is a high-level description of every public member function in each of the classes.

**StudentWorld**

StudentWorld(std::string assetPath)

This is the constructor and it takes the path to the assets folder.

virtual ~StudentWorld()

The destructor just calls the clean up function that will delete all of the dynamically allocated data.

virtual int init()

This function gets the world really to be played. It creates and places Socrates, all the pits, the dirt, and the food, returning a value to signal that the game is ready to be continued.

virtual int move()

This function is called very tick in the program. Socrates and every actor in the program is allowed to do something. Additionally, this function check if the game should continue and if new actors should be spawned.

virtual void cleanUp();

Deletes all the dynamically allocated data in the class.

void addToList(Actor\*);

This adds a new Actor to the vector of Actors. In this class because this class holds all actors in the game. Non-virtual because there is not child classes of StudentWorld.

bool collisions(Actor\* a, int dsitance, int amount);

This functions is used to see if a projectile runs into something that can be damaged, and damages it by the proper amount.

bool overLap(double x, double y);

Checks if the coordinates passed in overlap with a existing blockable actor. Non-virtual because no derived classes of StudnetWorld.

bool socratesOverlap(Actor\* a);

Checks if the passed in actor overlaps with Socrates. Non-virtual because no class inherits from StudnetWorld.

void setSocratesHealth();

Set Socrate’s health back to max. Non-virtual because no class is derived from StudentWorld.

void decreaseSocratesHealth(int decreaseAmount);

Decrease Socrate’s health by a certain amount. Non-virtual because no class inherits from StudentWorld.

void setSocratesFlameThrowerCharges();

Increases Socrates’s flame thrower charges. Non-virtual because no class inherits from StudentWorld.

bool edibleOverlap(Actor\* a);

Checks if passed in actor overlaps with edible actor. Non-virtual because no class inherits from StudentWorld.

bool blockChecking(double x, double y)

Checks if coordinates are withing blocking distance of a blockable actor. Non-virtual because no class inherits from StudentWorld.

void getCloseEdible(const double x, const double y, double& newX, double& newY)

Gets the coordinates of the closest edible actor in respect to passed in coordinates. Non-virtual because no class inherits from StudentWorld.

double socratesDistance(Actor\* a)

Finds the distance between actor and Socrates. Non-virtual because no class inherits from StudentWorld.

double targetAngle(Actor\* a);

Finds the angle an actor needs to travel to find Socrates. Non-virtual because no class inherits from StudentWorld.

**ACTOR**

Actor( int ID, double sX, double sY, Direction dir, int depth, double size, StudentWorld\* world);

This is non-default constructor for actor, contains basic attributes that all actors should have.

virtual void doSomething() = 0;

Function that is called every tick. This class is pure virtual because every class derived from actor needs to implement this function, and I do not want an instance of Actor to be created.

virtual ~Actor();

Destructor for Actor, does not contain anything. Virtual because their will be destructors in the derived classes.

bool getAlive();

Return if an actor is alive or not. Non-virtual because derived class because other classes should not have their own definition of alive.

void setAlive(bool x);

Sets actor to be alive or dead. Non-virtual because only actor has access to the m\_alive member variable.

StudentWorld\* getWorld();

Get the world the actor occupies. Non-virtual because only Actor has access to the pointer to the actor’s world.

virtual bool hitable() = 0;

Indicates if a actor can be hit by a projectile. Pure virtual because every derived class needs to state if it can be hit by a projectile.

virtual bool hasHitpoints() = 0;

Indicates if an actor has hit points. Pure virtual because every class needs to specify if it has hit points.

virtual bool blockable() = 0;

Indicates if an actor can block movement if another actor. Pure virtual because every class needs implement blockable.

virtual bool edible() = 0;

Indicates if an actor is edible. Pure virtual because every class needs to specify if it’s edible.

virtual void damage(int x) = 0;

Damages the actor. Pure virtual because every actor needs to specify, even though not all actors can be damaged.

**AGENT**

Agent(int ID, double sX, double sY, Direction dir, int depth, double size, StudentWorld\* sw, int hitPoints);

This is non-default constructor for Agent, like actor but also has hit points.

virtual ~Agent();

Destructor for agent. Virtual because classes are derived from it.

bool hitable();

Return true because agents are hitable. Non-virtual because this is true for all actors.

bool hasHitpoints();

Return true because agents have hit points. Non-virtual because this is true for all actors.

bool blockable();

Return true because agents are not blockable. Non-virtual because this is true for all actors.

int getHitPoints();

Return hit points. Non-virtual because only agent has access to hit points.

void setHitPoints(int x);

Set hit points. Non-virtual because only agent has access to hitpoints

bool edible();

Return false because agents are not edible. Non-virtual because this is false for all actors.

virtual void damage(int x) = 0;

Damage will damage every agent. Every agent needs to be damaged to it is pure-virtual. Also need to make Agent an abstract base class.

**Dirt**

Dirt(int ID, double sX, double sY, Direction dir, int depth, double size, StudentWorld\* sw);

Non-default constructor for dirt.

virtual ~Dirt();

Destructor for dirt. Virtual for coding consistency.

void doSomething();

Does nothing. Non-virtual because nothing inherits from dirt.

bool hitable();

Return true. Non-virtual because nothing inherits from dirt.

bool hasHitpoints();

Return false. Non-virtual because nothing inherits from dirt.

bool blockable();

Return true. Non-virtual because nothing inherits from dirt.

bool edible();

Return false. Non-virtual because nothing inherits from dirt.

void damage(int x);

Automatically kills dirt. Non-virtual because nothing inherits from dirt.

**PIT**

virtual ~Pit();

Destructor for pit. Virtual for coding consistency.

Pit(int ID, double sX, double sY, Direction dir, int depth, double size, StudentWorld\* sw);

Constructor for Pit.

void doSomething();

Will randomly spawn different types of bacteria until its quote is filled. Non-virtual because nothing inherits from pit.

bool hitable();

Returns false. Non-virtual because nothing inherits from pit.

bool hasHitpoints();

Returns false. Non-virtual because nothing inherits from pit.

bool blockable();

Returns false. Non-virtual because nothing inherits from pit.

bool edible();

Returns false. Non-virtual because nothing inherits from pit.

void damage(int x);

Empty, pit cannot be damaged. Non-virtual because nothing inherits from pit.

**SOCRATES**

Socrates(int ID, double sX, double sY, Direction dir, int depth, double size, StudentWorld\* sw, int hitPoints);

Constructor for Socrates, which is an agent.

void doSomething();

Depending on input from user. Socrates will shoot, move, or increase his spray. Non-virtual because nothing inherits from Socrates.

virtual ~Socrates();

Destructor for Socrates. Virtual for coding consistency

double getFlameThrowerCharges();

Returns flame charges for Socrates. Non-virtual because nothing inherits from Socrates.

void setFlameThrowerCharges(double x);

Sets flame charges for Socrates. Non-virtual because only Socrates has access to its flame charges

double getSprayCharges();

Returns flame spray for Socrates. Non-virtual because nothing inherits from Socrates.

void setSprayCharges(double x);

Sets spray charges for Socrates. Non-virtual nothing else has spray charges.

void damage(int x);

Damages Socrates and kill him if damage is too great. Non-virtual because nothing is derived from Socrates.

**PROJECTILE**

virtual ~Projectile();

Destructor for projectile, virtual for coding consistency.

Projectile(int ID, double sX, double sY, Direction dir, int depth, double size,

StudentWorld\* sw);

Constructor for projectile.

bool hitable();

Return false, non-virtual because this is the same for all projectiles.

bool hasHitpoints();

Return false, non-virtual because this is the same for all projectiles.

void doSomething();

Checks if projectile is dead and calls each projectile’s specialization function. Non-virtual because this contains similar behavior for every projectile.

bool blockable();

Return false, non-virtual because this is the same for all projectiles.

bool edible();

Return false, non-virtual because this is the same for all projectiles.

void damage(int x);

Return false, non-virtual because this is the same for all projectiles.

int getTravelDistance();

Returns how far projectile has traveled. Non-virtual because only Projectile has access to the travel distance

void setTravelDistance(int increaseAmount);

Increase travel distance by a certain amount. Non-virtual because only Projectile has access to the travel distance

virtual void specialization() = 0;

Special features of each derived class. Pure virtual because every derived class needs to have some special behavior.

**DISINFECTANTSPRAY**

void specialization();

This function sees if the spray collides with anything that is hittable and damages that actor by 2 hit points and dies. Or, spray moves into center of dish until it’s time has ended. This is non-virtual because nothing inherits from this class.

**FLAME**

void specialization();

This function sees if the flame collides with anything that is hittable and damages that actor by 5 hit points and dies. Or, spray moves into center of dish until its lifespan has ended. This is non-virtual because nothing inherits from this class.

**GOODIE**

void doSomething();

Checks if the goodie has overlapped wit Socrates, if so it calls a specialization function that performs the proper task. Or, the lifespan runs out. Non-virtual because this is necessary behavior for every derived class.

bool hitable();

Returns true. Non-virtual because this is similar for every derived class.

bool hasHitpoints();

Returns false. Non-virtual because this is similar for every derived class.

int getLife();

Returns lifespan left of goodie. Returns true. Non-virtual because this is similar for every derived class.

void decLife();

Decrease lifespan by one. Non-virtual because this is similar for every derived class.

bool blockable();

Returns false. Non-virtual because this is similar for every derived class.

bool edible();

Returns false. Non-virtual because this is similar for every derived class.

void damage(int x);

Kills goodie automatically. Non-virtual because this is similar for every derived class.

virtual void specialization() = 0;

Unique ability for every derived kind of goodie. Pure virtual since needs to be implemented by every derived goodie.

**RESTOREHEALTHGOODIE**

void specialization();

Increase score and restores Socrates’s health. Non-virtual because no class derives from RestoreHealthGoodie.

**FLAMETHROWERGOODIE**

void specialization();

Increase score and increases flame thrower charge. Non-virtual because no class derives from FlameThowerGoodie.

**EXTRALIFEGOODIE**

void specialization();

Increase score and increases amount of lives. Non-virtual because no class derives from ExtraLifeGoodie.

**FUNGUS**

void specialization();

Decrease score and health. Non-virtual because no class derives from Fungus.

**FOOD**void doSomething();

Does nothing. Non-virtual because nothing is based on Food.

bool hitable();

Returns false. Non-virtual because nothing is based on Food.

bool hasHitpoints();

Return false. Non-virtual because nothing is based on Food.

bool blockable();

Return false. Non-virtual because nothing is based on Food.

bool edible();

Returns true. Non-virtual because nothing is based on Food.

void damage(int x);

Empty, food cannot be damaged. Non-virtual because nothing is based on Food.

**BACTERIUM**

virtual void doSomething() = 0;

Each bacteria needs to do something a bit different. Pure virtual to make bacterium an abstract class, and every derived class needs to implement this.

int getMovementPlanDistance();

Return movementPlanDistance, non-virtual because child class should not override this function.

void setMovementPlantDistance(int x);

Set the movementPlanDistance, non-virtual because child class should not override this function.

int getFoodEaten();

Return the amount of food eaten, non-virtual because child class should not override this function.

void setFoodEaten(int x);

Set the amount of food eaten, non-virtual because child class should not override this function.

virtual void die() = 0;

Each bacterium will need to implement its own die function. Pure virtual because this needs to be implemented.

void damage(int x);

Bacteria will have it’s hit points damaged and if severe, it will die. Non-virtual because all bacterium are damaged the same way.

virtual void cry() = 0;

All bacteria need to implement a cry function that will play a sound if they are either hurt and die. Pure virtual because every kind of bacteria need implement this.

**REGULARSALMONELLA**

virtual void doSomething();

Will see if salmonella can split into two, or if the salmonella can eat food. Moreover, salmonella sees if it can damage Socrates. Salmonella then moves towards nearest edible actor and if it is blocked, it will move in random direction. Virtual because this function can be overridden by the AggressiveSalmonella class.

void die();

Will kill salmonella and increase score. Will play die sound and random chance to spawn food at that point. Non-virtual because this is used for AggressiveSalmonella as well.

void cry();

All Salmonella need to implement this function that will either play its hurt sound or die sound. Non-virtual because these sounds will be the same for all Salmonella.

**AGGRESSIVESALMONELLA**

virtual void doSomething();

Will see if salmonella can split into two, or if the salmonella can eat food. Moreover, salmonella sees if it can damage Socrates. Additionally, will check if Salmonella is in a proper area to move towards. Salmonella then moves and if it is blocked it will not move. If not in radius of Socrates, Salmonella will aim towards getting nearest edible actor. Non-virtual because nothing derives from AggressiveSalmonella.

void die();

Will kill salmonella and increase score. Will play die sound and random chance to spawn food at that point. Non-virtual because nothing derives from this class.

**ECOLI**

void doSomething();

Checks if it overlaps with Socrates, if so it damages Socrates. It will try to split, if not it will check if it has food to eat. Then it finds angle to Socrates and tries to navigate towards Socrates. If blocked will increment angle until it is free.

void die();

Will increase score and has 50% probability to spawn food at that location.

void cry();

All Ecoli need to implement this function that will either play its hurt sound or die sound. Non-virtual because nothing is based from Ecoli.

2. N/A

3. Here are assumptions I made

* The first assumption that I made was that the maximum amount of disinfectant was 20 charges, I got this assumption from the sample game
* That there is no limit on the amount of flame thrower charges Socrates can have.
* The maximum score that can be reached is 999999, while the lowest score one can have is 0
* I let each actor have access to the StudentWorld that they lived in
* I assumed that food could not be consumed by bacteria if that food was already dead
* Actors that are created during another Actor’s do something will have the chance to do something during the next call of the move function.
* I changed the disinfectant spray to spawn directly in front of the player to match the behavior on the sample program

4.

**StudentWorld**

I tested this class firstly by making sure that all the dynamic memory allocation was deleted since this class allocated a lot of data. I do not know how well I did this since there were already memory leaks in the graphics library. Additionally, to make sure that this class was stable at later levels, I wrote code to increase the level and made sure that game played out regularly. Additionally, for functions that relied on the presence of another actor, I made sure that the function only runs if that actor is still alive. Moreover, I made sure that memory we do not have access to was not accessed through making sure that the vector was properly iterated through.

**Actor**

This acts as the basis for every other class we made (not StudentWorld) so I cannot really test the code of this class. Specifically, the code of this class can only be tested through the implementation of the other classes. However, a check I did do was try to create a new Actor and make sure I got a compiler error.

**Dirt**

I tested the Dirt class by making sure that every kind of actor has the proper kind of behavior when coming across a dirt actor. For instance, Projectiles needs to die along with the Dirt. Also, every Bacterium needed to be blocked by the dirt.

**Agent**

This class is another class that is the basis for a lot of other classes. I ran a similar test as the Actor class since this class is also an abstract class.

**Pit**

I tested class through creating a game world that only consisted of a single pit. I made sure that the proper amount of bacteria were expelled at random intervals. Additionally, I made sure that the pit disappears as soon as all the bacteria are expelled. I then ran normal simulations making sure Pit was spawned in proper spots and released the correct amount of bacteria.

**Socrates**

I tested Socrates firstly by making sure that the movement was correct. I made sure that Socrates always when around the perimeter of the petri dish, even when Socrates goes around the petri dish multiple times in one direction. Furthermore, I made sure that all the keys the user can use causes Socrates to initiate the proper behavior, this involved making sure that the spray charges increased when no button is pressed. Also, I made sure that the amounts of both amounts of charges decreased and increased properly.

**Projectile**

This is another one of the abstract classes such as agent and actor. Therefore, direct testing was almost impossible. The testing for this class consisted of testing Flame and DisinfectantSpray. This checks the doSomething function which does what both projectiles have in common.

**Flame**

The way I tested Flame was making sure that Flames traveled in the correct direction. All the Flames had to move in the direction that they were spawned in. By not decreasing the amount of Socrates’s flame thrower charges, I was able to make sure that Flames moved the proper distance. The most important test I did was to make sure that Flames were set to dead once their maximum distance as reached. Also, I made sure that Flames automatically destroyed dirt and goodies, while damaging bacteria accordingly.

**DisinfectantSpray**

The way I tested Spray was making sure that Spray traveled in the correct direction. The spray had to travel towards the middle of the petri from any position from the perimeter. By not decreasing the amount of Socrates’s spray charges, I was able to make sure that Flames moved the proper distance. The most important test I did was to make sure that Spray were set to dead once their maximum distance as reached. Also, I made sure that Spray automatically destroyed dirt and goodies, while damaging bacteria accordingly.

**Goodie**

Goodie is another abstract class, which can be tested through its derived classes. This tested the doSomething function which is combined with each goodies’s specialization function.

**RestoreHealthGoodie**

Firstly, I made sure that this goodie spawned at its proper location which is around the perimeter of the petri dish. I made sure that the proper sprite was shown, along with that Socrates’s health was reset to max when overlapping with the goodie, and nothing bad happens if Socrates is already at 100 hit points. Additionally, I made sure that the score increased by the proper amount. Also, I ensured the projectiles killed the goodie.

**FlameThrowerCharges**

Firstly, I made sure that this goodie spawned at its proper location which is around the perimeter of the petri dish. I made sure that the proper sprite was shown, along with that Socrates’s flame thrower charges are incremented by one. Additionally, I made sure that the score increased by the proper amount. Also, I ensured the projectiles killed the goodie.

**ExtraLifeGoodie**

Firstly, I made sure that this goodie spawned at its proper location which is around the perimeter of the petri dish. I made sure that the proper sprite was shown, along with that another live was added to lives at the top of the screen Additionally, I made sure that the score increased by the proper amount. Also, I ensured the projectiles killed the goodie.

**Fungus**

Firstly, I made sure that the fungus goodie spawned at its proper location which is around the perimeter of the petri dish. I made sure that the proper sprite was shown, along with that Socrates’s health was decreased when overlapping with the fungus. Additionally, I made sure that the score decreased by the proper amount. Also, I ensured the projectiles killed the fungus.

**Food**

I tested the Food class by making sure that the food sprite was correct. Next, I made sure that Food was destructible by two kinds of projectiles, and Food will stay until it is eaten by bacteria. Additionally, I made sure that Food objects automatically were set to dead once they were consumed by bacteria.

**Bacterium**

Is the last abstract class in the program. The function that I could test in this class was the damage function that is shared by all bacteria. I tested this function through repetitively shooting bacteria to make sure that they are properly being damaged and calling their die functions once their hit points hit zero or below.

**RegularSalmonella**

The thing to check in this class was it’s doSomething function. The primary goal of regular salmonella is to eat all the food present on the petri dish, and then moving randomly after that. I created a world with a lot of food and made sure that regular salmonella went after and consumed all of the food and split properly. After this, the regular salmonella moves randomly. I then made sure that killing a regular salmonella increased the score properly.

**AggressiveSalmonella**

The thing to check in this class was its doSomething function. The primary goal of aggressive salmonella is to go after Socrates or to eat all the food present on the petri dish, and then moving randomly after that. I created a world with a lot of food and made sure that aggressive salmonella would go after Socrates if it was inside of its search area, if not the aggressive salmonella chooses to go find food. I then made sure that this class will not move if it is blocked from Socrates. Lastly, I made sure that killing a aggressive salmonella increased the score properly.

**Ecoli**

Testing the Ecoli was easier than the two kinds of salmonella. I created a world with a single Ecoli and made sure that it would always target Socrates. Moreover, I made sure that Ecoli will be blocked properly when placed in environment with Dirt. Lastly, I made sure that Ecoli can be split and increase the score the proper amount when killed.