Hanna Co

**Actor**

**double getDistance(double x1, double y1, double x2, double y2);**

* This function takes in four doubles, and computes the distance between the two points (x1, y1) and (x2, y2).

**bool isAlive();**

* This function returns the private data member m\_alive, which informs the StudentWorld whether or not the Actor is alive.

**virtual void doSomething() = 0;**

* This function is pure virtual because there is no default for doSomething, and this also prevents the user from creating an Actor object.

**StudentWorld\* getWorld();**

* This function returns the private data member m\_world, which is a pointer to a StudentWorld object. This allows the actor to call functions defined in the StudentWorld class.

**void setDead();**

* This function sets the private data member m\_dead to false, indicating the actor is dead.

**bool isDamageable();**

* This function returns the value of the private member variable m\_damage, which indicates whether or not the Actor can be damaged by flame or spray.

**bool canIncPoints();**

* This function returns the value of the private data member variable m\_incPoints, which indicates whether or not the StudentWorld object should increment the player’s points when an object is hit by a flame or spray.

**bool isEdible();**

* This function returns the value of the private data member m\_edible, which indicates whether or not the Actor is edible. This way, the StudentWorld can check if the bacteria is able to eat the object.

**bool blocks();**

* This function returns whether the value of the private data member m\_blocks, which returns whether or not the actor blocks the movement of bacteria. Thisway, the StudentWorld object can determine if the bacteria is able to move in a certain direction.

**int getHP();**

* This function returns the value of the private data member m\_hp, which is how many hit points the actor has. I made m\_hp a private data member of Actor, so that this function can be called in StudentWorld to access and decrement the HP. For actors where HP is not relevant, m\_hp is simply set to 1, and not accessed/changed in any function.

**void setHP(int amt);**

* This function sets the value of m\_hp to amt.

**virtual void playSpecificSound(bool isDead);**

* I created this function solely for the purpose of playing noise for when the bacteria is hit or dies, because salmonella and e.coli play a different noise. I didn’t make this function pure virtual because it was added later on, and didn’t need to be defined for any other actor. It’s virtual because it is overridden by the bacteria classes so that it plays the sound specific to the bacteria type, and depends on whether or not the bacteria is simply hurt, or is dead.

**Socrates**

**int getFlame();**

* This function gets the value of the private data member flame, which is how many flames Socrates has left.

**void setFlame(int amt);**

* This function sets the value of flame to amt.

**int getSpray();**

* This function returns the value of the private data member spray, which is how many sprays Socrates has.

**double calculate(Direction d, char axis);**

* This is a helper function that calculates new coordinates every time Socrates moves. It takes in Socrates’s current direction and calculates a new x or y coordinate, depending on which the user is trying to calculate.

**Attack**

**void doAttack();**

* This function first checks to see if the spray or flame is alive, returning immediately if it is not. Next, it calls the StudentWorld function damage, seeing if it overlaps with any objects it can damage. If it did not damage anything, it moves forward in its direction, and decreases its maxTravelDistance. If maxTravelDistance is 0, it is set to dead.

**int getMaxTravel();**

* This function returns the value of the private data member maxTravel, which is how many pixels that specific attack can travel.

**int getAmountDamage();**

* This function returns the value of the private data member amountDamage, which is much the object it overlaps with should decrease its hit points by.

**void setMaxTravel(int amt);**

* This function sets the value of maxTravel to amt.

**Goodie**

**void setLifeSpan(int amt);**

* This function sets the value of the private data member lifeSpan to amt.

**int getLifeSpan();**

* This function returns the value of the private data member lifeSpan, which is how many ticks the goodie is alive for,

**void commonGoodie(int ptInc);**

* This function performs all the common actions between the three goodies. It first decreases its life span by one, and checks if it is dead. If it is dead, it returns right away. It computes the distance between the goodie and Socrates, and if they overlap, it tells the StudentWorld object to increase the user’s points by a certain amount. It sets the goodie to dead, plays the correct sound, and calls the function goodieBonus, which is the differentiated stuff each goodie does. If its lifespan is 0, it sets the goodie to dead.

**virtual void goodieBonus() = 0;**

* This function does all the differentiated stuff each goodie does, which is why it is pure virtual. The user should not be able to create a Goodie object, and it needs to be overridden by each goodie. For example, HealthGoodie sets Socrates’s HP to 100, FlameGoodie gives Socrates 5 more flames, and LifeGoodie gives the user an extra life.

**Pit**

**int getRSal();**

* This function returns the value of the private data member rSal, which is how many regular salmonella the pit still contains.

**int getASal();**

* This function returns the value of the private data member aSal, which is how many aggressive salmonella the pit still contains.

**int getEColi();**

* This function returns the value of the private data member eColi, which is how many E. Coli the pit still contains.

**void decRSal();**

* This function decreases the value of the private data member rSal by one, indicating that it has released a bacteria of that type.

**void decASal();**

* This function decreases the value of the private data member aSal by one, indicating that it has released a bacteria of that type.

**void decEColi();**

* This function decreases the value of the private data member eColi by one, indicating that it has released a bacteria of that type.

**Bacteria**

**void setMPD(int amt);**

* This function sets the value of the private data member movePlanDist to amt, indicating the bacterium has plans to move.

**int getMPD();**

* This function returns the value of the private data member movePlanDist.

**int getEaten();**

* This function returns the value of the private data member eaten, which indicates how many food objects the bacteria has eaten.

**void setEaten(int amt);**

* This function sets the value of the private data member eaten to amt.

**void attackSocrates(int damageAmt);**

* This function decreases Scorates’s HP by damageAmt, which is how much damage each type of bacteria does. If Socrates’s HP is less than or equal to zero, play the sound to indicate he has died, and set Socrates to dead. Otherwise, play the sound to indicate he was hurt.

**double multiplyX();**

* This function calculates a new X coordinate for when the bacteria multiplies, adding or subtracting SPRITE\_WIDTH / 2 depending on its distance from the center of the petri dish.

**double multiplyY();**

* This function calculates a new y coordinate for when the bacteria multiplies, adding or subtracting SPRITE\_WIDTH / 2 depending on its distance from the center of the petri dish.

**void salmonMove(Direction dir);**

* This function first decreases movePlanDist by 1, and calculates a new X and Y coordinate 3 pixels in the salmonella’s direction. If moving there is not possible, with the salmonella either overlapping with dirt or moving outside the petri dish, it gives it a random direction and resets movePlanDist to 10. Otherwise, it will move to the new x and y coordinate.

**void eatFood(Bacteria\* cur);**

* If food is found within 128 pixels. the Direction variable food is set to the direction of the food, and the salmonella tries to move to the food, eating it if they overlap. If no food is found, it is given a random direction, movePlanDist is reset to 10, and the function returns.

**StudentWorld**

**bool checkOverlap(Actor\* cur, int limit);**

* This function iterates through the vector of actors up to limit, seeing if cur overlaps with any of them. If it doesn’t, it returns false.

**Socrates\* getSoc();**

* This function returns m\_soc, which is a private data member which a pointer to Socrates.

**Actor\* getActor(int index);**

* This function returns the actor in m\_actors at index.

**bool addFungus();**

* This calculates the chance of a fungus appearing in the petri dish, returning true if a Fungus should be added, and false if it should not.

**bool addGoodie();**

* This function calculates the chance of a goodie appearing in the petri dish, returning true if a goodie should be added, and false if it should not be.

**void addBacteria(Pit\* cur, double x, double y);**

* This function adds bacteria from the pit. It creates a vector of options, pushing a number that corresponds to the bacteria type, but only if the pit still contains bacteria of that type, It then randomly generates an index and returns the number at the index, indicating what type of bacteria to add. It then creates a new bacteria of that type, adds it to m\_actors, increases m\_numBacteria, and decreases the number of that type of bacteria in the pit.

**void addBacteria(Actor\* cur);**

* addBacteria is an overloaded function, so this version is called when a Bacteria multiplies as a result of eating three food objects. It adds cur to m\_actors, and increases m\_numBacteria by one.

**void addSpray(double x, double y, Direction start);**

* This function creates a new Spray object, adding it to m\_actors.

**void addFlame(double x, double y, Direction start);**

* This function creates a new Flame object, adding it to m\_actors.

**void addFood(double x, double y);**

* This function creates a new Food object, and adds it to m\_actors.

**bool damage(double x, double y, int amt);**

* This function iterates through m\_actors, calculating the distance between (x, y) and the current actors location. If they overlap and the actor is able to be damaged by flame or spray, then it decreases the actors hp by amt. If you should increase the user’s points, then do so. If the actor is dead, set it to dead, and play the correct sound. If it is bacteria (indicated by whether or not you should increase points) and it died and need to become a food object, then add a new food object,

**bool blocked(double x, double y, double distance);**

* This function iterates through m\_actors, and if the distance between (x, y) and the current actor’s location is <= distance, and the current object blocks movement, then return true. Else, return false.

**bool becomeFood();**

* This function calculates the chance of bacteria becoming food once it dies, returning true if the bacteria should become a food object.

**bool eat(double x, double y);**

* This function iterates through m\_actors, and sees if the bacteria overlaps with any edible objects. If so, it sets the food object to dead and returns true. Otherwise, it returns false.

**bool findFood(Actor\* cur, Direction &dir);**

* This function iterates through m\_actors, finding the closest edible actor within 128 pixels. If there is none, it returns false. Else, it computes the angle between the closest food object and (x, y), setting dir to that angle, and returning true.

**void deletePit();**

* This function decreases the private data member m\_numPits by one, indicating that the pit has emitted all its bacteria.

**void deleteDead();**

* This function iterates through m\_actors, checking to see if it is dead. If it is, it is deleted from m\_actors. If it is bacteria (indicated by whether or not you should increase points), then decrease m\_numBacteria, indicating there is one less bacteria in the petri dish.

**bool findSocrates(Actor\* cur, Direction &dir, int howFar);**

* This function calculates the distance between cur and Socrates. If the distance between them is less than or equal to howFar, it set dir to the angle between cur and Socrates, and return true. Else. is Socrates is too far, return false.

1. For Socrates, it says “Socrates must wait one or more ticks without spraying to replenish his spray charges”. It was unclear whether or not this meant Socrates can’t have sprayed the tick before, or if it meant that the sprays are only replenished if no key was pressed that tick. I interpreted it to mean the latter, so if the player does not press a key for that tick, then Socrates’s sprays increase by one. For actors such as spray and flame, where the spec says they only damage one object it overlaps with, I chose this simply to be whichever Actor comes first in the vector.

For the majority of the classes, it was easier to observe it during gameplay, to make sure it behaved properly.

**Socrates**

To test that Socrates moved properly, I just moved him around in circles, making sure his distance from the edge of the petri dish remains constant. Testing the user’s input was also fairly simple: I just made sure the right projectile was fired, and that Socrates moved in the right direction. If I press the enter key five times, Socrates runs out of flames, which is correct. Making sure he had enough sprays was more difficult, as they pretty much never run out (seeing as they regenerate multiple times a second), but the text at the top displays the correct number of sprays.

**Regular Salmonella**

I couldn’t think of a way to test this function, but it appears to move randomly around the petri dish, and moves towards food when it is within 128 pixels. It dies when it overlaps with a projectile, and does damage to Socrates, so it appears to function normally. It does multiply after eating three food objects, and about half of the time, it becomes a food object when killed.

**Aggressive Salmonella**

The only difference between the aggressive and regular salmonella is that the aggressive salmonella will move towards Socrates if the distance between them is 72 pixels or less, so I simply tested this by moving Socrates close to an aggressive salmonella, and seeing if it would go after Socrates.

**E. Coli**

E. coli was fairly easy to test, as its only action is to go after Socrates. Right when it’s added to the petri dish, it goes after socrates. When Socrates would change direction, the e. coli changed its direction as well to move towards Socrates. When it is killed, sometimes it becomes a food object, and sometimes it simply disappears, which is what it should do.

**Pits**

To make sure it emitted the correct number of bacteria, I counted the total number of bacteria emitted, which was ten: two e. coli and eight salmonella. The order of which they were emitted also varies, so I assume my method of choosing a type of bacteria was correct.

**Flames**

It was hard for me to tell if flames traveled as far as they should, but when compared to the version of Kontagion posted, it looked to be about the same distance. The flames would appear when the enter key was pressed, and appeared in a full circle around Socrates. When it overlapped with bacteria or dirt, the object it overlapped with would die. To make sure the flames were delivering enough damage, I commented out the code that moved the bacteria, and counted how many times a bacteria would have to be hit before it died.

**Sprays**

As with the flame, it’s hard to tell exactly if the spray travels far enough, but when compared to the full Kontagion version uploaded, they appeared to travel a similar distance. To test whether it delivered damage, I did the same thing I did with flame, where I stopped the bacteria from moving, and counted how many times it took to kill the bacteria.

**Dirt**

This class was a bit difficult to test, as they don’t do much, but whenever a dirt object was hit by a spray or a flame, it would disappear. Additionally, during gameplay, it was possible for bacteria to get stuck behind dirt, indicating that it correctly blocks the bacteria’s movement.

**Food**

As with dirt, the food class was a bit difficult to test because of its simplicity, so I mostly observed its behavior during gameplay. When bacteria overlapped with it, it would disappear. Additionally, flames and spray did not damage it, indicating that it behaved as it should.

**Restore Health Goodies**

Whenever a health goodie appeared, I would simply move Socrates so that they overlapped. If Socrates’s hp increased to 100, then I knew it was working properly. I also made sure the user’s points would increase by the correct amount.

**Flamethrower Goodies**

Similar to the health goodie, I would move Socrates to overlap with it when it appeared, and see if the flame count displayed at the top of the screen increased by five. I also kept track of Socrates’s points to make sure they increased by the correct amount.

**Extra Life Goodies**

Just like the other two goodies, I simply moved Socrates so that he overlapped with it, and made sure that the “lives” count at the top of the screen increased by one. I also watched to make sure the player’s score increased by the correct amount.

**Fungi**

If the fungus overlapped with Socrates, I watched and made sure Socrates took enough damage, and that the user lost the correct amount of points. If it overlapped with a flame, the fungus would disappear. It’s supposed to do the same when it overlaps with spray, but I wasn’t sure how to test this, so I assumed it behaves as it should.

**StudentWorld**

I tested the init function by making sure it added enough dirt and food objects, as well as the correct amount of pits. Socrates always began at the left of the petri dish, which he was supposed to. With every new level, Socrates’s sprays, flames, and hit points reset correctly. None of the pits, foods, or dirts were too close to the edge of the petri dish, and no food objects overlapped with pits or dirt, and vice versa. Dirts did occasionally overlap with each other, but that was allowed. I tested my move function by seeing that the number of sprays increased every tick, and that every actor did something each tick. Every dead actor disappeared, and when all pits and bacteria were gone, the player advanced onto the next level. I tested for every situation: running out of lives and quitting the game, dying but having lives left, and finishing the game. The cleanup function was tested when I compiled the game on g32, and didn’t get any memory leaks.