Project 3 Report

1. Each of the classes below that is a base class has a virtual destructor. Although none of the classes allocated memory for their data members, I defined all of them out of convention. Typically, you want your base class to have a virtual destructor so that it can call all subclasses’ destructors when the base class object is destroyed.

Additionally, each of the classes below has a public constructor that I did not describe because I thought describing each individual constructor’s functionality and reason why I defined it in its respective class would be trivial

**Actor**

I chose to define a pure virtual version of the doSomething() function in my base Actor class because all actors in Kontagion are able to do something, and each type of actor does something in a different way.

I chose to define a pure virtual version of the activate() function in my base Actor class because all actors in Kontagion affect other actors when overlapped, and each type of actor affects in a different way.

I chose to define a pure virtual version of the affectHp() function in my base Actor class because I needed all the bacteria classes and the Socrates class to have their hp changed, and each class changed hp in a different way. I made the implementation of affectHp() do nothing in classes that don’t have hp.

The takeDamage() function is used by bacteria objects and the Socrates object. This is why I defined it in the actor class. This function reduces hp and kills actors if necessary based on if they are damageable or not.

The getWorld() function returns a pointer to the StudentWorld the game is in. It is necessary in most classes that need to access game data, so I put in in the actor class.

The getAlive() function returns a boolean telling if the actor is alive or not. It is used by many of the derived classes and also by the StudentWorld class to tell whether an actor should be affected or make an action. This is why I put it in the actor class.

The getSolid() function returns a boolean telling if the actor can be passed through or not. It is used by many of the derived classes and also by the StudentWorld class to tell whether an actor can move to a certain space. This is why I put it in the actor class.

The getEdible() function returns a boolean telling if the actor can be eaten by bacteria or not. It is used by the bacteria subclasses and also by the StudentWorld class to tell whether an actor should be eaten. This is why I put it in the actor class.

The setAlive() function sets m\_alive to true or false depending on if the actor is dead. I put it in the actor class because almost every derived class has some functionality that needs to set an actor to dead (especially in the activate() function).

**Socrates**

The doSomething() function takes input from the user and either moves Socrates, shoots spray or flames, or replenishes spray depending on the key pressed. This is the base class, so it had to be defined here.

The activate() function does nothing since nothing happens to another actor when it touches Socrates.This is the base class, so it had to be defined here.

The affectHp() function works with the takeDamage() function to handle the specific hp losses required for the Socrates object. This is the base class, so it had to be defined here.

The getSpray() function returns the amount of spray left. This is the base class, so it had to be defined here.

The getFlame() function returns the amount of flames left. This is the base class, so it had to be defined here.

The getHp() function returns the amount of hp left. This is the base class, so it had to be defined here.

The setFlame() function changes the amount of flames left. This is the base class, so it had to be defined here.

The setHp() function changes the amount of hp left. This is the base class, so it had to be defined here.

**Dirt**

The doSomething() function does nothing for dirt. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Dirt can be instantiated.

The activate() function does nothing for dirt. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Dirt can be instantiated.

The affectHp() function does nothing for dirt. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Dirt can be instantiated.

**Pit**

The doSomething() function randomly spawns one of the three types of bacteria once every 50 ticks on average. It plays the bacteria birth sound, initializes a new bacterium at its own position, and adds a pointer to the bacteria to the ActorList. I defined it here because it is the base class.

The activate() function does nothing for pit. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Pit can be instantiated.

The affectHp() function does nothing for pit. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Pit can be instantiated.

**Food**

The doSomething() function does nothing for food. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Food can be instantiated.

The activate() function does nothing for food. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Food can be instantiated.

The affectHp() function does nothing for food. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Food can be instantiated.

**Bacteria**

The doSomething() function calls the pure virtual specificBehavior() function that varies according to the type of bacteria in action. Depending on the passed by reference boolean in specificBehavior, the function may also move the bacteria according to its movement plan or set a new movement plan if there is not one already. This function is not virtual because all the specific functionality is handled by specificBehavior(). I defined it here because it does all the things in common between the three types of bacteria each tick.

The activate() function either eats or damages the Actor it is in contact with depending on if it is food or Socrates, respectively. I defined it here because every bacteria does these things, the only difference is how much Socrates is damages, which is handled by the pure virtual damageSocrates() function.

The affectHp() function lowers the hp of the bacteria by the set amount. If the new hp is less than or equal to zero, the function kills the bacterium and has a 50% chance to spawn a new food object at the position occupied by the bacterium. I defined it here because every bacterium behaves the same when it is damaged.

**Regular**

The only public function in this class is the constructor. It is defined here because every class needs a constructor.

**Aggressive**

The only public function in this class is the constructor. It is defined here because every class needs a constructor.

**EColi**

The only public function in this class is the constructor. It is defined here because every class needs a constructor.

**Goodie**

The doSomething() function decreases lifetime each tick and if lifetime is 0, it kills the goodie. I defined it here because each type of goodie behaves the same way each tick.

The activate() function is pure virtual in Goodie. I defined it as such here because Goodie cannot be instantiated and each specific type of goodie does something different when activated.

The affectHp() function does nothing for Goodies. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Goodie subclasses can be instantiated.

**ExtraLife**

The activate() function increases Socrates lives by 1 and score by 500. I defined it here because this is the base class.

**RestoreHealth**

The activate() function sets Socrates’ hp to 100 and increases score by 250. I defined it here because this is the base class.

**RestoreFlame**

The activate() function increases Socrates’ number of flames by 5 and increases score by 300. I defined it here because this is the base class.

**Fungus**

The activate() function decreases Socrates’ hp by 20 and his score by 50. I defined it here because this is the base class.

**Weapon**

The doSomething() method moves the weapon forward and increases travelDistance by the same amount. It uses the pure virtual function checkDead() to handle the specific cases of sprays and flames where they are set to dead after travelling a certain distance. I defined it here because there is shared functionality between spray and flame each tick.

The activate() function is pure virtual because each weapon does something different when it is overlapped and because weapon itself never needs to be instantiated.

The affectHp() function does nothing for Weapon. I needed to define it here because it is pure virtual in Actor and needed to be redefined if Weapon subclasses can be instantiated.

**Flame**

The checkDead() function checks if travelDistance is greater than or equal to 32, and if so sets it to dead. I defined it here because flame has different death requirements than spray.

The activate() function calls takeDamage() to deal 5 hp of damage to the object overlapping with the flame. I defined it here because this is the base class.

**Spray**

The checkDead() function checks if travelDistance is greater than or equal to 112, and if so sets it to dead. I defined it here because spray has different death requirements than flame.

The activate() function calls takeDamage() to deal 2 hp of damage to the object overlapping with the flame. I defined it here because this is the base class.

**StudentWorld**

The constructor initializes the member variables player, nBacteriaLeft, and levelOver.

The destructor just calls cleanUp() which will be explained later.

The init() function populates the petri dish with dirt, food, and pits, as well as initializing the Socrates object at the left edge of the dish. It also ensures that non-dirt objects do not overlap. Finally, it makes sure that all of these new actors are added to ActorList.

The move() function first allows the player to doSomething(), then it iterates through ActorList and calls doSomething() on each actor. It then reiterates through and calls activate() on each actor to determine what happens if two objects overlap. It then reiterates a third time, deleting dead actors and their pointers from ActorList. The function also randomly spawns goodies and fungus around the map, adding them to ActorList as well as updates the score.

The cleanUp() function iterates through ActorList and deletes all the actors as well as Socrates.

The getPlayer() function returns a pointer to Socrates.

The addActor() function takes in a pointer to an actor and adds it to the end of ActorList.

The increaseBacteria() and reduceBacteria() functions increment and decrease the nBacteriaLeft count that allows us to tell when the level is over.

The openSpace() method takes in an x and y coordinate, iterates through ActorList, and returns true if no other actors in the list are overlapping with that coordinate.

The findDirToFood() function takes in a coordinate and returns the angle to the nearest food object if it is within 128 pixels of the coordinate. If no such food object is found, it returns a random angle.

The findAngleBetweenPoints() method takes in two coordinates and uses the first one as the vertex and finds the angle to the second one (with east being 0).

The overlap() method takes in a pointer to an actor and first checks to see if it overlaps with player. It then iterates through ActorList and checks for overlaps. The function returns a pointer to the first object it finds that overlaps or nullptr if no such actor is found.

The distance() method takes in two coordinates and returns the distance between them.

1. I implemented all the classes, but there were a lot of bugs that I could not figure out how to fix. The game itself is slow and will often stops working. I suspect this is an issue with birthing and killing objects and how they are removed or added to the actor lists. Looking back on it, I shouldn’t have separated functionality into doSomething() and activate(), but rather mak them one function, that way I wouldn’t need to have two separate for loops in my move() class. Additionally, the spray animations do not show up on the screen and the flame animations do not move forward. Both classes do damage and play their respective sounds though. I also noticed that fungi and goodies do not spawn. Perhaps this is due to the same reason (ActorList not being able to handle new objects being created while it is still iterating through).
2. For the bacteria doSomething() function, the spec said to choose a random direction as the movementPlanAngle if the previous movementPlanAngle would cause the bacteria to be blocked by dirt. I did this, but it causes the bacteria to get stuck in areas where there’s a lot of dirt because these random angles will often lead to more blocks. In the sample game, the bacteria do not get stuck like this, but I did not know why.

The spec did not really give much information regarding what to do when more than two objects occupy the same space. In situations like this, I made sure that if Socrates was in the same position as something that would damage him, he would be damaged. Otherwise, the first actor in ActorList would be the one affected due to the nature of the overlap() function returning the first actor it found at that position.

In the spec, for the lifetime of a fungus, it said 300 –10 \* L -1 or 50, whichever was higher. Since the first term had no parenthesis, I assumed it meant 300 - (10 \* (L - 1)).

The spec did not specify if goodie perks carried over between levels. For example, if a restoreFlame goodie made gave the player 8 flames, I did not know if in the next level the player should have 8 flames or 5. I decided to reset all statistics between levels (ie. the player would start every level with 5 flames).

1. **Socrates**

I tested Socrates by moving him in a full circle around the dish to make sure that the angle would reset to 0 after 359. I also tried holding down the left and right keys to make sure that Socrates would move smoothly between ticks and that his direction would always be facing the middle. I also made sure to try holding down the spray and fire to make sure the counts would decrease to 0 and not go any further. Additionally, I let him get killed by the bacteria to test that he would take damage, lose lives, and eventually end the game.

**Dirt**

I tested the dirt by shooting it with flames and sprays to make sure it would disappear. I also checked to make sure bacteria did not go through it. Additionally, I checked to make sure that no new dirt objects were created in the game after initialization.

**Pit**

I tested the pit by waiting and making sure that all three types of bacteria were spawned. I also decreased the counts of each type to 0 and made sure that the pit did not spawn any bacteria in this situation. I also tried shooting the pit and making sure that it did not take damage or die. Lastly, I observed the pit during the course of the game to make sure that bacteria were not blocked by it.

**Food**

I tested the food object by setting the divide() function so that bacteria would only need to eat once to divide. This ensured that the foodEaten count would increase when the bacteria overlapped with the food. Since Food’s methods didn’t do anything, I couldn’t really test it by manipulating its own code, so I just observed it during the game to make sure it would actually disappear when in contact with bacteria.

**Regular**

Before testing regular, I set its image ID to Socrates so I could distinguish it from the aggressive salmonella. I tested the regular class by killing a regular salmonella and making sure that it died after being hit by two sprays or one flame. I ensured that the death sound and hurt sounds played but that the hurt sound did not play once the regular salmonella died. I made sure that the divide() function worked by lowering the amount of food needed and seeing what happened when it ate. I also allowed it to damage Socrates and made sure it did the appropriate amount of damage per tick. I also tried to make sure that the doSomething() function was working by seeing if it would get stuck in area with lots of dirt. Unfortunately all of my bacteria objects would get stuck in these situations and I did not have enough time to figure out why.

**Aggressive**

My tests for aggressive were similar to my tests for regular. However, in my tests of this class, I made sure that these bacteria would follow my Socrates object at the right radius. To do this, I created a petri dish with only a Socrates and an aggressive salmonella and I spawned them 73 pixels away and noted to make sure the bacteria wouldn’t always travel toward Socrates. However, when I spawned them 71 pixels away from each other, I made sure that the salmonella would always be following Socrates.

**EColi**

My tests for EColi were similar to my tests for regular except for the fact that I ensured the EColi would always be following Socrates no matter where he was on the map. Because the distance cap for following and the diameter of the dish were both 256, the EColi should always be following Socrates.

**ExtraLife**

As mentioned above, I couldn’t get any of the goodies to work or show up on the screen. I suspect that this is an error of the StudentWorld class and not of the derived classes themselves though. If I could get them to work, I would have tested the extra life goodie by making sure it died when shot. I also would have obviously tried collecting them and making sure it actually incremented Socrates’ lives and disappeared after collection. I would have also changed the chances of spawning to 100% and 0% to make sure that the percent chance actually was working. Lastly, I would ensure that the extra life did not carry over between levels.

**RestoreHealth**

My tests of restore health would have been identical to my tests of ExtraLife. I would additionally have checked the my health would be restored to 100 hp exactly, not more and not less. I would have also made sure that when a bacterium overlapped with it that its health did not increase.

**RestoreFlame**

Once again, these tests would be identical to ExtraLife with the addition of testing to make sure the flame count was incremented by 5 and not just reset to 5. I would also check that the flames did not carry over between levels.

**Fungus**

Again, these tests would be identical to above with the exception that I would check that the player’s hp would be reduced by 20 in case of contact and that the fungus would disappear after contact.

**Flame**

I tested the flame by spawning it in an empty petri dish and making sure it died after 32 pixels. I also made sure to count that there were 16 flames in a charge after Socrates shot them. Additionally I checked to make sure that flames would disappear after hitting objects and that these objects were actually damaged. I made sure that flames damaged bacteria, dirt, and goodies/fungi, but not Socrates.

**Spray**

I tested the spray by spawning it in an empty petri dish and making sure it died after 112 pixels. I also made sure that the spray animations didn’t glitch when shot continuously. Additionally I checked to make sure that sprays would disappear after hitting objects and that these objects were actually damaged. I made sure that sprays damaged bacteria, dirt, and goodies/fungi, but not Socrates.