Report, Project 3

Abhishek Marda

The program implemented everything required by the spec. No assumptions were made; everything in the code has been implemented as directed by the project spec.

**Actor.h and Actor.cpp**

Actor base class:

* Actor(**int** imageID, **int** startCol, **int** startRow, Direction facing, **int** depth, StudentWorld\* world) : GraphObject(imageID, startCol, startRow, facing, depth){};
  + Initializes the actor base object by setting it’s status to alive and setting the StudentWorld pointer which is then called getStudentWorldPtr()
* StudentWorld\* getStudentWorldPtr();
  + Returns the pointer to the associated StudentWorld class.
  + Not declared virtual since this remains constant for all for all of the classes.
* **virtual** **void** doSomething()=0;
  + Declared pure virtual since every class compulsorily should have a doSomething() function described for it.
* **bool** isAlive();
  + Returns true if the object is alive, else returns false.
  + Not declared virtual since this remains constant for all for all of the classes.
* **virtual** **bool** isDamageableByProjectile();
  + Determines if an object can be damaged by a projectile (flame or spray).
  + By default, this returns true since many of the objects can be damaged by a projectile. However, this is declared virtual and can be redefined by objects such as pits which cannot be damaged by a projectile.
* **virtual** **bool** isDamageableByBacteria();
  + Determines if an object can be damaged by a bacteria object.
  + By default, this returns false since many of the objects cannot be damaged by a bacterium. However, this is declared virtual and can be redefined by objects such as food and Socrates which can be damaged by a bacterium.
* **virtual** **bool** isObstacle();
  + Determines if an object is an obstacle, i.e., it hinders movement.
  + By default, it returns false. However, it is declared virtual for objects that can block bacteria (i.e. dirt).
* **void** setDead();
  + Sets an objects status as dead.
  + Not set as virtual since the implementation does not change across objects.
* **virtual** **void** damage(**int** amount);
  + Damages an object by a certain amount.
  + By default it sets an object dead, however it is declared virtual for objects that have health such as bacteria and Socrates which decrement the health and will die only when their health is less than or equal to 0.
* **virtual** ~Actor(){}
  + The destructor is declared virtual to avoid memory leaks, so that the base class’ destructor calls the destructor of the derived classes.

Socrates class:

No functions are kept virtual in this class since no class is further derived from it. The only ones marked virtual are for good programming practice.

* Socrates(StudentWorld\* world) : Actor(IID\_PLAYER, 0, VIEW\_HEIGHT/2, 0, 0, world){};
  + Initializes the Socrates’ health to 100 and its inventory to 20 sprays and 5 flame charges.
* **virtual** **void** doSomething();
  + Provided the player is alive, this function gets an input from the user and based on the input, it will move right or left, fire a disinfectant spray, or fire a flame thrower charge. If nothing is being done, the number of spray charges is refilled.
* **virtual** **bool** isDamageableByBacteria();
  + Redefines the function from the base class to return true.
* **virtual** **void** damage(**int** amount);
  + Damage the player by specified amount. If the health is less than or equal to 0, the player is set dead.
* **void** setHealth(**int** hp);
  + Set health to specified amount.
  + The function ensures that the health is set between 0 and 100, and if 0, the player is set dead.
* **int** getHealth();
  + Returns the health of the player.
* **void** flamethrowerRecharge();
  + Adds 5 flame thrower charges to the total count of flame throwers remaining. This function is called when Socrates lands on a flame thrower goodie.
* **int** getSprays();
  + Returns the number of sprays left.
* **int** getFlameThrowerCharges();
  + Returns the number of flame throwers left.

Projectile class:

Two classes are further derived from this class: Spray and Flame. The derived classes have no functions; they only have constructors which also provide the Projectile base class with the damage the projectile does and the maximum distance available which the projectile class stores in a private data member.

* Projectile(**int** imageID, **int** col, **int** row, Direction dir, StudentWorld\* world, **int** maxDistance, **int** damage) : Actor (imageID, col, row, dir, 1, world){};
  + Initializes the maximum distance, its current distance travelled, and the damage that it would create when it overlaps with something that can be damaged by a projectile (which is passed into the Projectile object base class from its derived classes).
* **virtual** **bool** isDamageableByProjectile();
  + Overrides the Actor class’ implementation and returns false. A projectile cannot be harmed by another projectile.
* **virtual** **void** doSomething();
  + Provided the player is alive, if there is overlap of the projectile with an object that can be damaged by a projectile, then the damage is done, and the projectile is set dead along with it.
  + If there isn’t any such overlap, then the function advances the position of the projectile.
* **void** moveForward();
  + Moves the projectile forward as long as it has not crossed its maximum available distance. If it has, then the projectile is set dead.

Dirt class:

No functions are kept virtual in this class since no class is further derived from it. The only ones marked virtual are for good programming practice.

* **virtual** **bool** isObstacle();
  + Since dirt can block a bacterium’s movement, this function overrides Actor class’ definition and returns true.
* **virtual** **void** doSomething();
  + Does nothing.

Food class:

No functions are kept virtual in this class since no class is further derived from it. The only ones marked virtual are for good programming practice.

* **virtual** **void** doSomething();
  + Doesn’t do anything
* **virtual** **bool** isDamageableByProjectile();
  + Overrides the Actor class’ definition to return false. Foods cannot be damaged by projectiles.
* **virtual** **bool** isDamageableByBacteria();
  + Overrides the Actor class’ definition to return true. Foods can be damaged by bacteria.

Goodies class:

Four classes are derived from this class: RestoreHealth, FlameThrowerCharge, ExtraLife, Fungus.

Each of these classes return the amount of damage and lifetime (in number of ticks) to the Goodies class in their respective constructors.

* Goodies (**int** image, **int** col, **int** row, StudentWorld\* world, **int** lifetime, **int** points) : Actor (image, col, row, 0, 1, world){};
  + Initializes the Goodies’ lifetime and points gained when the player overlaps with it (which is passed to the goodies base class from its derived classes constructors)
* **virtual** **void** doSomething();
  + Provided the object is alive, if there is overlap of the object with Socrates, then the player’s score is updated accordingly and the specific ability of each goodie type item is called. If there is not overlap, then the lifetime of the object is decremented and if it becomes less than 0, the status of the object is set dead.
* **virtual** **void** doAbility()=0;
  + This function calls on the specific ability of every derived object from Goodies class. It is defined as pure virtual since every object derived from Goodies class should have its own doAbility() function. The individual implementation for every class of doAbility() is listed below
    - RestoreHealth: set player’s health to 100
    - FlameThrowerCharge: add 5 flame thrower charges to player’s inventory
    - ExtraLife: increase the number of lives left
    - Fungus: decrease the player’s health by 20 points

Bacteria class:

There are three classes derived from this class: RegularSalmonella, AggressiveSalmonella, and EColi. The constructors of each of these classes return their initial health, the damage they do, and the points the player earns when it successfully kills that type of bacteria.

* Bacteria(**int** image, **int** col, **int** row, StudentWorld\* world, **int** health, **int** damage, **int** killpoints) : Actor(image, col, row, 90, 0, world){};
  + Initializes the bacteria’s health, amount of damage it causes to the player, the amount of food eaten initially, movement plan distance, the points gained when the player kills the bacteria, and increments the number of bacteria remaining in the petri dish in the StudentWorld. The health, damage cause, and points gained are passed from its derived classes.
* **virtual** **void** doSomething();
  + Provided the bacteria is alive, the function checks whether there is a player is within a 72 pixel radius (only in the case of aggressive salmonella; no such check will happen for regular salmonella or ecoli).   
    This function will then check if there is overlap with Socrates and will damage him accordingly. If there is overlap with a food object, then it will set the food object’s status to dead and will divide if required.   
    Lastly, the function will call the changeMovement() function which is described below.
* **virtual** **void** damage(**int** amount);
  + Decrements the health of the bacteria by the provided amount. If the bacteria is to be killed, the chance of it becoming a food object is also calculated and implemented accordingly.
* **void** divide();
  + This function is called when a bacteria has eaten 3 food objects and has to create a new instance of itself. This function gets appropriate coordinates for the new objects and calls the duplicate() function with those coordinates which creates the appropriate object. The implementation of duplicate() is described below.
* **virtual** **void** duplicate(**int** col, **int** row)=0;
  + This function creates a new bacterium of the same type that calls it at the provided coordinates. It is different and has to be declared for every type of bacterium, thus it is declared pure virtual. For regular salmonella, it creates a regular salmonella, for aggressive salmonella, it creates another aggressive salmonella and for ecoli it creates another ecoli.
* **bool** tryToMoveInDir (**int** px, **int** overlapRadius=SPRITE\_WIDTH/2);
  + If it can move in the current direction (i.e it is in bounds and not blocked by a dirt pile) of the bacterium for the specified number of units, it will do so and return true, otherwise it will return false without making any change.
  + It is not change across derived classes since there is nothing about this function that is dependent to the type of the bacteria, thus it is not declared virtual.
* **bool** targetNearestFood (**int** radius);
  + Finds the nearest food object within the specified radius, set the direction to that food object and return true. If there is no object within the specified radius, then return false.
  + It is not change across derived classes since there is nothing about this function that is dependent to the type of the bacteria, thus it is not declared virtual.
* **bool** targetSocrates (**int** radius);
  + If there is Socrates within the specified radius, then set the direction of the bacteria toward Socrates and return true. If there isn’t Socrates in the specified radius, then return false.
  + It is not change across derived classes since there is nothing about this function that is dependent to the type of the bacteria, thus it is not declared virtual.
* **virtual** **bool** becomeHeatseeker();
  + The purpose of this function is for aggressive salmonella to ascertain whether there is Socrates in 72 px, and if there is, then move toward Socrates. By default, this function returns false.
  + Since this function changes for Aggressive salmonella, the function has to be virtual.
* **virtual** **void** changeMovement();
  + This function directs how the bacteria will move for the current tick. By default its implementation works for regular and aggressive salmonella however it is declared virtual since it changes for ecoli.
    - By default: if the bacteria doesn’t need the change direction, it won’t and will continue in its current trajectory provided it is not blocked. If it needs to check direction, it will get the nearest food source within 128 pixels and move in the direction of that food object. If there is no food object in that radius, a random direction will be chosen.
    - For ecoli: it finds Socrates’ position provided it is within a radius of 256 pixels. If it is, then ecoli will try to move in that direction provided it is not blocked. If it is blocked, then for upto 10 tries, it will change its direction and try to move again.
* **void** setRandomDirection();
  + Set the direction to a random number between 0 and 359 degrees.
  + This implementation does not change across classes, thus it is not declared virtual.
* **void** setMovementPlan(**int** newplan);
  + Set the movement plan to the specified amount.
* **int** getMovementPlan ();
  + Get the movement plan amount.
* **virtual** **void** hurtSound();
  + Plays the sound when the bacteria is hurt.
  + By default it plays the sound associated with either salmonella, however it changes for ecoli and is thus declared virtual.
* **virtual** **void** deadSound();
  + Plays the sound when the bacteria is killed.
  + By default it plays the sound associated with either salmonella, however it changes for ecoli and is thus declared virtual.

Pit class:

No functions are kept virtual in this class since no class is further derived from it. The only ones marked virtual are for good programming practice.

* Pit(**int** col, **int** row, StudentWorld\* world) : Actor (IID\_PIT, col, row, 0, 1, world)
  + Initializes the inventory of the pit with the number of every type of bacteria.
* **void** releaseRegularSalmonella();
  + Creates and initializes a regular salmonella.
* **void** releaseAggressiveSalmonella();
  + Creates and initializes an aggressive salmonella.
* **void** releaseEColi();
  + Creates and initializes an ecoli.
* **void** releaseRandomBacteria();
  + Randomly chooses which bacteria to initialize.
* **bool** empty();
  + Returns true if the inventory of the pit is empty.
* **virtual** **bool** isDamageableByProjectile();
  + Overrides the Actor class’ implementation to return false since pits cannot be destroyed by a projectile.
* **virtual** **void** doSomething();
  + Sets the pit’s status to dead if it is empty, otherwise it has a 1/50 chance of releasing a bacteria.

**StudentWorld.h and StudentWorld.cpp**

* int init()
  + Initializes all of the pits, foods, and dirt objects. It creates the Socrates object and adds all of these objects to the petri dish.
* int move()
  + Makes every object that exists in the container of actors do its specific behavior. If the player dies, it ends the game, if the level is won, it goes to the next level, otherwise it continues the game. If there is an object that has been killed, then it is removed from the game. It also adds the different types of goodies or fungi, if required.
* void cleanup()
  + Deletes all of the dynamically allocated objects.
* **void** add(Actor\* a);
  + Add the pointer passed in to the vector that contains pointers to all of the actors.
* **bool** isOverlap (Actor \*a1, Actor \*a2, **int** radius =SPRITE\_WIDTH);
  + Check if there is a minimum distance of specified radius between the two actor objects.
* **bool** hasAnyOverlap (Actor \*a, **int** noOfStartingElements, **int** radius = SPRITE\_WIDTH);
  + Find if there are any overlapping objects of passed-in actor for the starting noOfStartingElements elements in the vector that contains all of the actors where the maximum distance of overlap should be radius pixels.
* **bool** hasAnyOverlap (Actor \*a);
  + Find if there are any overlapping objects of provided actor for the entirety of the vector where overlap is defined as the max distance SPRITE\_WIDTH.
* **void** getRandomPosition(**int** &x, **int** &y);
  + Get random position for pits, food, and dirt within 120 pixels from the center.
* **bool** projectileDamageableOverlap (Actor \*a, **int** amount=0);
  + Damages one object that can be damaged by a projectile and is overlapping with a projectile.
* **bool** levelWon();
  + Returns true if the number of bacteria and pits left in the game are both 0.
* **void** addNoOfBacteria(**int** n);
  + Increments n number of bacteria to the data member that holds the number of bacterium.
* **void** decrementPits();
  + Reduce the number of pits remaining (stored in a private data member) by 1.
* **void** setPlayerHealth(**int** hp);
  + Set the player’s health to provided amount.
* **void** pickUpFlamethrowerCharge();
  + Increment 5 flame thrower charges to the player.
* **int** getPlayerHealth();
  + Return the player’s current health.
* **bool** bacteriaDamageableOverlap (Actor \*a);
  + Damages one object that can be damaged by a bacteria and is overlapping with a bacterium.
* **bool** bacteriaIsBlocked (Actor \*a, **int** radius);
  + Returns true if the Actor is overlapping with a dirt pile implying that the actor is blocked, else returns false. This function is only called within bacteria.
* **bool** getNearestFood (Actor \*a, **int**& x, **int** &y, **int** radius);
  + Find the nearest food item in the provided radius to the actor and modify x and y accordingly and return true, and if no such food item exists then make no changes and return false.
* **void** socratesPos(**int** &x, **int** &y);
  + Get Socrates’ position
* **void** damagePlayer(**int** damage);
  + Damage the player by the specified amount.

**Testing**

A major part of the testing was done by frame-by-frame analysis of my implementation compared with the provided executable file. The analysis of every class through this method is specified under its header below.

**Actor**

Actor class gets tested automatically as other functions work. Since there is no specific image for actor class that gets displayed on the screen, it is hard to test it individually, however the successful implementation of its derived classes implies that it is working.

**Socrates**

Socrates class has to be tested in multiple ways. First, we have to check whether it after being constructed at the right position on the screen it is able to move left or right. Since Socrates moves 5 degrees in every key press of left or right, we can say that if it moves from the leftmost position to the top in 18 clicks, the player has correct translational motion. The rotation of Socrates too must be taken into account since as it moves along the perimeter of the Petri dish it should always point to the center. Every time Socrates is hurt is should generate the related sound. Next one has to check the implantation of spray and flames. Keeping space bar pressed to deploy the spray which would eventually get its count to 0, after which sprays should stop firing. Releasing the space bar should allow the sprays to start replenishing. Similarly, flames should appear in a circle and not fire more than 5 times after pressing the enter key provided no goodie for flame thrower was consumed. Sound of spray and flames should also be played every time the appropriate key is pressed. Pressing f and spectating frame by frame, spray should travel for 14 ticks and flames should persist for 4 ticks. On the death of Socrates or when it has successfully finished a level, the correct sound should be played.

**Flame**

Checking of whether flame is being correctly moved and created is covered above in the testing of Socrates. However, along with that flames should be able to remove dirt piles and kill bacteria. They should also be able to remove goodies that appear at the perimeter. Apart from that, it cannot damage anything else and cannot be damaged by anything except for itself. It should also be seen whether in the same tick that if the flame overlaps something it can damage, it also damages itself. Seeing that the right amount of damage is being done by the flame object can be done by ensuring that aggressive salmonella takes 2 hits to die by flame however regular salmonella dies after coming in contact with just a single flame object.

**Spray**

Checking of whether spray is being correctly moved and created is covered above in the testing of Socrates. However, along with that sprays should be able to remove dirt piles and kill bacteria. Apart from that, it can not damage anything else. It should also be seen whether in the same tick that the spray overlaps something it can damage, it also damages itself. Seeing that the right amount of damage is being done by the spray object can be done by ensuring that aggressive salmonella takes 5 hits to die by spray however regular salmonella dies after coming in contact with just a single spray object.

**Dirt**

Dirt piles should be initialized randomly in the screen and initially not overlap with anything but themselves. When hit by a projectile they should disappear along with the projectile that hit them. They should successfully be able to block a bacteria. They should also not be created further that 120 pixels from the center.

It is also important to note the number of spray items being generated which can be done by hard coding the number of food items to a small number. Making the game simpler by reducing the number of bacteria would help ensure that the correct number of dirt items are being created at the right distance. Trying to localize the radius of creation of dirt objects and placing a food or pit in between can help detect whether the dirt objects successfully don’t overlap with a food or pit.

**Food**

Food objects should be initialized randomly in the screen and initially not overlap with anything. When overlapping with a bacteria they should disappear along with the bacteria. They should also not be created further that 120 pixels from the center. They should not be damaged by projectiles, which should instead ensure that they go over them. They should be able to attract salmonella and consumption of 3 of them should make the specific bacteria divide itself. It is also important to note the number of food items being generated which can be done by hard coding the number of food items to a small number. Making the game simpler by reducing the number of bacteria and dirt piles would help ensure that the correct number of food items are being created. Trying to localize the radius of creation of food objects and placing a pit in between can help detect whether the food objects successfully don’t overlap with a pit.

**All types of Goodies**

They should appear on the screen only at the perimeter of the petri dish. For all goodies not fungi, they should play the “got goodie” sound when consumed by the player and for fungi it should play “player hurt” or “player dead” sound. Making the game simpler by decreasing the dirt piles and food and number of bacteria would also help to see how the lifetime and probability of creation changes over time to ensure that the current level’s difficulty is also being accounted for. They should only be destroyed if the player moves onto them, their lifetime ends, or they were killed by a flame object. These objects were also tested by normally using their implementation of their interaction with Socrates. Their effect on the screen should also be noticed: it should be noted whether the points are being incremented correctly for the score and for every object, that object is doing its thing properly. For example, ExtraLife should grant an extra life to the player and Fungus should damage the player.

**Pit**

The number of pits for every level should be counted for that level and it be ensured that during creation there is nothing overlapping with pits. The pits should only disappear after all of the bacteria are out. It would be easier to test with cout statements every time a bacteria is released to ensure that the pit is being killed at the right time. It should be ensured that a pit cannot be destroyed by bacteria or projectile. The only way it should be able to die is when all of the bacteria are created.

**Regular Salmonella and Aggressive Salmonella**

Removing dirt piles and keeping only food objects helps track whether the salmonella is moving in the right direction, and whether once all the food objects are destroyed, the food objects move in random direction. Then adding dirt piles and seeing that the salmonella changes direction on coming in contact with one should be tested. Frame by frame comparison with the provided exe file can help ascertain that the movement is being changed when the overlap between dirt and salmonella is SPRITE\_WIDTH/2. For regular salmonella, keeping a food object in front of Socrates to bait the salmonella and then firing projectiles one by one helps ensure that the salmonella is being damaged properly. To test an aggressive salmonella, cout statements can reveal the distance between the player and salmonella, and frame by frame analysis can ensure that the salmonella becomes aggressive when the distance of it with the player is 72 pixels, and then firing at it can ensure that it has the correct health and damage being dealt to it. For both of the salmonella, keeping only food objects can ensure that they are dividing when they eat 3 food objects. Frame by frame analysis can show that the correct damage is being done to the player when the salmonellas overlap with it. The sounds should be checked too for each of the following cases: a salmonella is created, hurt, or killed, and when it overlaps with the player. There should be an observation of the fact that a salmonella may become a food object on its death. Lastly, it should be seen that the correct points are given to the player when it dies.

**Ecoli**

Removing all obstacles and creating an ecoli should ensure that the ecoli heads directly to the player. Then adding obstacles would make sure that the ecoli successfully tries to move direction around it. Analyzing frame by frame helps ensure that a projectile does correct damage do it. Then when there is overlap with the player, it should be ensured that the correct amount of damage is being done every frame. Sounds of it being created, hurt, or killed as well as when it damages the player should be checked. There should be an observation of the fact that an ecoli may become a food object on its death. Lastly, it should be seen that the correct points are given to the player when it dies.

**StudentWorld**

If all of the classes are working properly and are being displayed on the screen well as required by the expected behavior, then it is safe to assume that the StudentWorld class is working properly.