1) High Level Description of Public Member Functions

Actor:

virtual void doSomething() – I made this virtual, but not pure virtual because objects derived from the Actor class like the dirt object and food object do something in similar, nothing. Therefore, I just implemented it so it would automatically be called for their doSomething().

StudentWorld\* getAPointerToMyStudentWorld()- It just returns a pointer to the StudentWorld that actors are part of, this is in the base class because all Actors need to know which StudentWorld they are part of, not virtual because just one function can return

bool isAlive() – returns whether an Actor is alive, implemented in base class because all Actors are either alive or not, which is handy in the base class and not in a virtual member function

void setDead()- public member function that allows other functions to set Actor to dead

bool damageable()- some objects, such as dirt are not damageable, so know whether to reduce HP, in a base class so that all derived classes can make use of it

virtual void impacted( int impact\_damage) – this serves as the HP modifier for all actors, subtracting damage heals, also can control sounds played and setDead if an actor dies

bool edible() – returns whether this actors is a food, useful for later bacterium to determine whether to consume a food, therefore this should just be in Actor and not anywhere else

bool blocksObjects() – returns whether this actor is a dirt object, useful for later bacterium to determine whether it is blocked by dirt, therefore should be in Actor and not elsewhere

virtual void playHurtSound() – this is virtual because some Actors do not produce a sound if hurt, or just cannot be hurt, therefore the base virtual is just no sound played, implement this function if want a sound produced

virtual void playDieSound() – this is virtual because some Actors do not produce a sound if they die, therefore the base virtual is no sound played, implement this function if want a sound produced if an Actor dies

int HP() – returns HP because all Actors have hitpoints, therefore no need for this to be implemented virtual, but rather in the base class

Socrates:

void doSomething() – this function checks for a key pressed, and reacts accordingly:

A left key or right key is moving counterclockwise or clockwise

Pressing space sprays a charge, and expends one if spray charges are greater than 0

Pressing enter sprays flamethrower charges, if they are above 0

int FlamethrowerCharges() – this returns Socrates flamethrower charges so that the StudentWorld can access it for the status game text

int SprayCharges() – this returns Socrates spray charges so that the StudentWorld can access it for the status game text

void PlayerMove(double angle\_change) – moves counterclockwise or clockwise, depending on the parameters passed, also adjusts the setDirection to the positional angle + 180, where positional angle is determined by incrementing the angle\_change, it moves in the sin(positional\_angle), cos(positional\_angle) for the y and x values, respectively

void getProjectileTrajectory(double& X, double& Y, Direction angle) – Returns where the projectile is going, specific to Socrates so that it can calculate whether it will overlap someday

void replenishFlameThrower() – add flamethrower charges so that the flamethrower goodie can easily adjust Socrates’ goodie

void playHurtSound() – Socrates plays a hurt sound if hit, so play it

void playDiesSound() – Socrates play a die sound if hit, unlike other actors, so play it

Dirt:

doesn’t have any member functions, does nothing

Projectile:

implement a base class for Flamethrower and Spray charges

void doSomething() –

return if not alive

see if it overlaps with another object (goodie, dirt, bacterium)

move in the direction it is facing

Spray:

makes use of the ammo member variable of Projectile

Flame:

makes use of the ammo member variable of Projectile

I implemented Spray and Flame together in a Projectile class because both function similarly in their movement, just differed in how many ammo and the max distance it can travel

Food:

no member functions like dirt, do nothing

Goodie:

base class:

bool steppedOnGoodie() -

see if Socrates is overlapping

set itself as dead

play sound that Goodie has a member variable of

void doSomething() – do the specific requirements of Goodie derived classes, this is a base class function because all Goodies do the same thing when stepped on, and play their respective sound when pressed on:

call steppedOnGoodie

if the lifetime is over, set itself as dead, call lifetime\_over for this

bool lifetime\_over() – if lifetime is over, set itself as dead

HealthGoodie, FlamethrowerGoodie, ExtraLifeGoodie, and Fungus all make use of the Goodie base class because they all have similar effects when overlapped, which makes it convenient to have a base class for them

Bacterium:

int foodConsumed() – returns the foodEaten by the bacterium, all bacteriums eat food, makes this function useful

void resetFoodEaten()- allows all Bacterium derived functions to reset food eaten

void increaseFoodEaten() – allows all Bacterium to increase the food eaten if they ate a food

bool socratesOverlap() – returns whether Socrates is overlapped

calls a StudentWorld function to accomplish this

double returnDirection() – returns direction of an actor to another actor, useful for a bacterium to move to another

bool calculateConsumed() – this function handles if the food consumed is 3, which creates a new Bacterium based on the derived class calling it

void spawnFood() – determines whether to spawn a food (50% chance)

bool detectBlocking(int pixels) –

check object’s x and y

compare with direction pixel x and y

if there is dirt, or edge of petri dish

return that there is indeed a blockage

I used Bacterium as a base class because salmonella and bacterium have similar functions and how they move. They also both consume food and make use of Socrates checking if it is overlapped.

Salmonella:

void playHurtSound() – overrides because Salmonella have a unique hurt sound

void playDieSound() – overrides because Salmonella have a unique die sound

void setMovementPlan(int movementplan) – sets the new movement plan

void newMovementPlan() – resets the class movement plan

int movementPlanValue() – returns the movement plan

I created this base class for Salmonella because both the aggressive and regular Salmonella have similar functions, IID, movement plan, and the sounds they produce when overlapping

Regular Salmonella:

void repopulate(double x, double y) – overrides so that calculateConsumed() produces specific object at regular salmonella

void doSomething()

if not alive, stop

if overlapping with Socrates

skip to step 5

if consumed 3 food objects

skip to step 5

if overlapping with a food

increase food eaten

if movement plan is greater than 0

see if still can move without blocking

if can’t then have a new movement plan

else move in that direction

if movement plan is less than 0, then find nearest food object

if no near food object, have a new movement plan

if moving to the food object is blocked, have a new movement plan, otherwise just move in the direction

Aggressive Salmonella:

void repopulate(double x, double y) – overrides so that specific object produced at aggressive salmonella

void doSomething()

if not alive, stop

try to find Socrates in 72 pixels

if did find, try to move in the direction

if blocked, do nothing

now only do steps 3-5

if overlapping with Socrates

then skip to step 6

see if consumed 3 food objects

otherwise see if overlapping with a food object

increase food eaten

if it has a movement plan

decrement movement plan

if blocked, reset movement plan

else move in that direction

otherwise

try to find food object close by

if no food object, then reset movement plan

if there is

try to move there

otherwise have a new movement plan

I actually could a doSomething for both the Reg Salmonella and Aggressive Salmonella, but because of the differing steps and checks, I decided against it. However, they do share a base class of Salmonella because they both utilize a movement plan, just do it at different times. They both also seek food and consume it similarly. However, EColi is much more different and doesn’t utilize the same movement plan but rather just uses less algorithms.

E Coli:

void repopulate(double x, double y) – adds a new Actor specific to E Coli

void doSomething() –

if not alive, stop

if overlapping with Socrates, skip to step 5

see if consumed 3 foods

else see if overlapping with food

try to find Socrates in 256 pixels

if he is nearby, try to move there

if can’t then move 10 degrees 10 times

stay still if can’t move

Pit:

int bacterium\_in\_stash() –

return how many bacterium still not spawned

void doSomething() -

1/50 chance to spawn bacterium

pick one of 3 bacterium accordingly, evenly

notify StudentWorld that a new bacterium is spawned

StudentWorld:

int init() – calls spawnObjects to spawn dirt, food, and pits

int move() – calls each Actors doSomething and Socrates doSomething

deletes dead actors

void cleanup() – unallocate all data members of Actors in StudentWorld

void addActor(Actor\* actor) – add actor to vector, be careful if used while iterating

void increaseBacterium(int howmany) – how many new Bacterium are introduced

void increasePitsThatSpawnedAllBacterium() – see if the pits that used all bacterium is same as level, which means all bacteria spawned

void SocratesImpactHP(int hp) – affect Socrates HP

StudentWorld Algorithms

bool overlap\_condition(double x, double y, Actor\* a, int limit)

Pythagorean formula using the double x, y and Actor a’s coordinates if less than the radius of limit

I use this overlap\_condition to check for overlap

bool checkForOverlap(double x, double y, bool checkingSpawnOverlap)

if not checking spawn overlap

check overlap

then for each actor if damageable, damage one actor

if checking spawn,

just make sure no overlap

bool overlapWithFoodOrDirt(double, double y, bool TestFood)

if testing food, overlap condition with SPRITE\_WIDTH

if not testing food, overlap condition with SPRITE\_WIDTH/2

Actor\* findFoodObjectInRange(double x, double y)

find FoodObject in 128 pixel range

return nullptr if there is no such foodobject

Actor\* findSocrates(double x, double y, double range)

find Socrates in range

return nullptr if Socrates isn’t there

bool isNotValidRangeFromObject()

use overlap condition to test against the range

bool ExactRangeFromObject()

is similar, but instead checks if exactly the range

void validSpawnSpot()

check that there is no overlap for this spawn

int move()

socrates do something

all actors do something

delete dead actors

set game text using scores, level … etc

2) Implementation Problems

I believed I managed to correctly implement all the classes.

However, one potential bug is sometimes my bacterium (E Coli and Aggressive Salmonella) get stuck on the Petri walls if Socrates doesn’t move. I’m not sure if this is intended behavior.

3) Assumptions

I made the assumption that initializing the world was easier by starting with the dirt class, then food, then pit. The spec lists it in the reverse order, but since dirt can overlap with itself, it is easier to do that first, then make sure no more objects overlap with each other while initializing. That was the only real assumption that I blindly made, because the effect was essentially the same. Most of the project I implemented by just following the spec word for word.

4) Testing My Classes

For the Actor class, testing any class basically could test this derived class, because all the graphics and whatnot are stored here. However, I still did test this by having it have different depths and IIDs to see whether my construction was appropriate. Furthermore, I checked that the HP values and impact were all working properly.

For the Socrates class, I had an empty petri dish except for dirt. I then fired my spray projectiles to see if they went away, along with any dirt, as well as with my flamethrower. I also made sure that when holding my space key, it worked properly as it did in the example Kontagion program. Furthermore, to test that Bacterium died in the amount of shots they did, I used my phone (kind of ghetto) to slow mo to make sure they died in appropriate ticks.

For the Spray class, derived from Projectile, I used my above explanation to ensure that the charges disappeared properly. Also I tried to slow mo how far the projectile went in an effort to see if they properly stopped, to somewhat success. I did the same for the flamethrower, but also counting the flame projectiles to be 16, and somewhat evenly spread.

For the Dirt class, I had only bacterium and dirt, and saw that the dirt blocked the bacterium. Also, I saw that the dirt disappeared upon contact with projectiles. I also counted the dirt objects after initialization to ensure that the proper amount were initialized.

For the Food class, I made sure that they disappeared when overlapped with bacterium. I also only had bacterium and food, and saw that the bacterium correctly reproduced when they ate 3 food objects. I was also observant that they sought the food objects if I wasn’t nearby, as with the movement algorithm.

For the HealthGoodie class derived rom Goodie, I listened for the sound to ensure it was properly played (and not fungus) which was also derived from Goodie. I also only had goodies in the petri dish and picked them up to see the score increase and also saw that it increased appropriately, along with making sure that my HP was back to 100 by taking a little damage from bacterium.

I did the same tests with the FlamethrowerGoodie, making sure the sound was played, score was given, and that I got 5 flamethrower charges. I also attacked the flamethrower goodie with my flamethrower to make sure that it disappeared. I essentially did the same for the ExtraLifeGoodie, making sure that I got an extra life.

The fungus was different, I made sure I got subtracted points, no sound was played, but it still disappeared when I hit it with a projectile.

For the Pit class, I made Socrates immortal and just counted how many bacterium spawned from the Pit. I made sure it was correct. I also made sure it disappeared after all the bacterium were released.

Bacterium:

For the Salmonella (regular), I made sure that it didn’t follow me, but rather food objects. I did this by not initializing anything other than food and me and a pit. I saw that the Salmonella ate up the food and spawned another bacterium if it ate 3. Also, next I spawned dirt to make sure that the Salmonella were blocked.

For the aggressive Salmonella, it was similar but I made sure that it followed me over food objects. I did the same by having an almost petri dish except for food, me and the Salmonella. I saw that the Salmonella did indeed seek the food, but ultimately me first If I was in range. I also noted that the Salmonella could get stuck on dirt blocks, something that regular Salmonella couldn’t do.

For the E Coli, I knew the range of it was 256 pixels, which was basically the entire Petri dish. I only had it with me in the Petri dish and had it follow me. I also noted that It tracked me counterclockwise, which is correct because it goes in 10 degree increments if it can’t reach me. I also noted that it could get stuck like the aggressive Salmonella.

The StudentWorld class was basically used by me every test. All my functions used in actors were mostly used through StudentWorld to check for overlap and such. Therefore, if something failed it was probably an algorithm problem in the StudentWorld class. One initial problem I had was that adding actors while moving would reallocate my vector’s memory, causing undefined behavior. I thus had it so that new Actors added during execution were only processed after the initial tick that they spawned it. This was an except for the projectiles fired from Socrates, because he fired them before the actors had a chance to doSomething, so no new allocations were added. The algorithms used in my StudentWorld were also shown in effect when proper overlaps were displayed, and no object went outside of the Petri dish.