Implementations of Actor Classes

# Actor Implementation

Actor serves as the base class (directly or indirectly) for all actors.

1. **Constructor:** *Actor(int imageID, double startX, double startY, StudentWorld\* world, Direction dir = 0, int depth = 0, double size = 1.0)*
   1. On the basis of imageID, the constructor body also decides whether or not the actor is damageable
   2. The member initialisation list of the constructor passes in the appropriate parameters to the base *GraphObject* object
   3. All actors are created alive
2. **Mutators**
   1. ***doSomething***
      1. Declared as pure virtual, since all the actors do something different
      2. It could make sense to use a non-pure virtual implementation where the default function does nothing; for clarity’s sake, however, I decided to make it pure virtual and write empty *doSomething* member functions for Dirt Piles and Food objects, so anyone looking at those classes would see they do nothing
   2. ***setActorDead***
      1. By default, it sets the actor’s state to dead (as tracked by a private data member), since there’s nothing else associated with setting most objects’ states to dead
      2. Declared as virtual, since *Bacterium* objects (derived from Actor) not only set the object state to dead, but also notify StudentWorld the bacterium died
   3. ***damageActor***
      1. Declared as virtual; the default implementation sets actor to dead
3. **Accessors (all constant)**: Each actor must be able to provide a pointer to its StudentWorld container, and must be able to say whether its whether they’re alive, damageable, eatable, or block movement
   1. ***getWorld***: returns a pointer to the StudentWorld object that contains the actor; non-virtual, since it works the same for all actors
   2. ***actorAlive***: returns a boolean representing whether the actor is alive; non-virtual, since it works the same for all actors
   3. ***actorDamageable***: returns a boolean representing whether the actor is damageable (true for bacteria, goodies, fungus, and dirt piles)
      1. Virtual function that returns false by default; the *Dirt Pile* class and *Goodie* and *Bacterium* base classes redefine it to return true
   4. ***actorEatable***: returns a boolean representing whether or not a bacteria is eatable (only true for Food objects)
      1. Virtual function that returns false by default; the *Food* class redefines it to return true
   5. ***actorBlocksMovement***: returns a boolean representing whether or not the actor blocks movement (only true for Dirt objects)
      1. Virtual function that returns false by default; the *Dirt Pile* class redefines it to return true

# Socrates Implementation

Socrates is derived from the Actor base class.

1. **Constructor**: *Socrates(StudentWorld\* world)*
   1. Creates a Socrates object at position (0, VIEW\_HEIGHT / 2), and the appropriate number of sprays, flame charges and hitpoints
2. **Mutators**
   1. ***doSomething***
      1. Returns if Socrates is dead
      2. If Socrates isn’t dead, checks for user’s key press
         1. If the user pressed a directional key, adjusts Socrates position accordingly
         2. If the player pressed space/enter, creates a new Spray/Flame object in the appropriate position using StudentWorld’s *createProjectile* template function and plays SOUND\_PLAYER\_SPRAY/SOUND\_PLAYER\_FIRE
         3. If player didn’t press a key, increments number of sprays Socrates has and returns
   2. ***damageActor***: Damages Socrates by decreasing hit points; sets Socrates to dead if hit points reach zero, and plays SOUND\_PLAYER\_HURT or SOUND\_PLAYER\_DIE as appropriate
      1. Socrates naturally has a different damage implementation than any other actor, and therefore gets its own specialised function
   3. ***restoreHitPoints***: restores Socrates hit points to full (used when Socrates picks up a Restore Health goodie)
   4. ***increaseFlames***: function that increases the number of flame charges Socrates has as specified by parameter (used when Socrates picks up a Flamethrower Charge goodie)
3. **Accessors** **(all constant and non-virtual)**
   1. ***getSprays***, ***getFlames****,* ***getHitPoints***: constant functions that return the number of spray charges/flame charges/hitpoints Socrates has left (used by *doSomething,* for displaying the status line, et cetera)

# Dirt Pile Implementation

The Dirt Pile class is derived from the Actor base class.

1. **Constructor**: *DirtPile(double startX, double startY, StudentWorld\* world)*
   1. Creates a dirt pile at specified coordinates
2. **Mutators**
   1. ***doSomething***: does nothing, since dirt piles do nothing
3. **Accessors** (both virtual and constant)
   1. ***actorsBlocksMovement***: returns true, since a dirt pile blocks movement
      1. Specialisation of Actor bas class’s *actorBlocksMovement* function, which returns false by default
   2. ***actorDamageable***: returns true, since a dirt pile is damageable
      1. Specialisation of Actor bas class’s *actorDamageable* function, which returns false by default

# Food Implementation

The Food class is derived from the Actor base class.

1. **Constructor**: *Food(double startX, double startY, StudentWorld\* world)*
   1. Creates a Food object at specified coordinates
2. **Mutators**
   1. ***doSomething***: virtual function; does nothing, since food objects do nothing
3. **Accessors** (both virtual and constant)
   1. ***actorEatable***: returns true, since a food object is eatable
      1. Specialisation of Actor bas class’s *actorEatable* function, which returns false by default

# Pit Implementation

The Pit class is derived from the Actor base class

1. **Constructor**: *Pit(double startX, double startY, StudentWorld\* world)*
   1. Creates a Pit object at specified position, and sets the number of regular/aggressive salmonella and E. Coli in the pit to the default numbers per spec
2. **Mutators**
   1. ***doSomething***
      1. If the pit is all out of bacteria, sets pit to dead and returns
      2. If pit isn’t out of bacteria, pit has 1/50 chance of creating a bacterium; simulates this chance by creating a bacterium only if *randInt(1, 50) == 1*
      3. Repeatedly, until we create a bacterium:
         1. Generate a random number *rand* between 1 and 3, inclusive
         2. If *rand* is 1 and pit hasn’t run out of Regular Salmonella
            1. Create a new Regular Salmonella using StudentWorld’s template *createBacterium* function
         3. If *rand* is 2 and we pit hasn’t out of Aggressive Salmonella
            1. Create a new Aggressive Salmonella using StudentWorld’s template *createBacterium* function
         4. If *rand* is 1 and we haven’t run out of Regular Salmonella
            1. Create a new Regular Salmonella using StudentWorld’s template *createBacterium* function
      4. Play bacterium creation sound

# Projectile Implementation

The Projectile class is derived from Actor and acts as a base class for the Spray and Flame classes. Since flames and sprays do very similar things (travel in a straight path, damage objects and die as a result, et cetera), it makes sense to have a base class for them.

1. **Constructor**: *Projectile(int imageID, double startX, double startY, Direction dir, StudentWorld\* world)*
   1. Appropriately constructs base Actor object with specified direction and coordinates, depth 1
   2. In constructor body, a switch statement evaluates the image ID and to decide what the initial travelable distance (tracked by a private data member should be) – 112 for a Spray, 32 for a Flame
2. **Mutators**
   1. ***doSomething***: All projectiles do the same thing:
      1. Uses StudentWorld’s *findAndDamage* function to see whether the projectile overlaps with a damageable object; if it does, the *findAndDamage* function damages one such object (with the appropriate amount to of damage, as given by *getDamage*) and returns true
      2. If projectile damaged an object, sets the projectile to dead and returns
      3. Otherwise, finds the next coordinates in its path, moves to those coordinates, and decrements the travelable distance as appropriate
      4. If remaining travelable distance is less than zero, sets projectile to dead
   2. ***decTravelableDist***: Decreases travelable distance of projectile by specified amount
3. **Accessors**
   1. ***getDamage***: pure virtual function; it’s defined by Spray and Flame to return the damage each kind of projectile deals
   2. ***getTravelableDist***: returns the distance the projectile can still travel

# Flame and Spray Implementations

Flame and Spray classes are derived from Projectile.

1. **Constructor**: *Flame/Spray(double startX, double startY, Direction dir, StudentWorld\* world)*
   1. In member initialisation list, we construct the Projectile base class with appropriate parameters
2. **Accessor**: ***getDamage***
   1. Virtual function that returns the amount of damage dealt by the spray/flame; i.e. 2 for spray, 5 for flame

# Goodie Implementation

Goodie is derived from Actor and serves as a base class for the RestoreHealthGoodie, FlamethrowerGoodie, ExtraLifeGoodie, and Fungus classes. Since each of these objects do very similar things (i.e. check for overlap with Socrates and affect Socrates in some way if yes), it makes sense to have a base class for them.

1. **Constructor**: *Goodie(int imageID, double startX, double startY, StudentWorld\* world)*
   1. Creates an Actor at the specified position with direction 0 and depth 1
   2. Sets life of the goodie as specified in spec
2. **Mutators**
   1. ***doSomething***: All goodies do the same thing:
      1. If goodie isn’t alive, return immediately
      2. If goodie overlaps with Socrates
         1. Update Socrates score as appropriate with GameWorld’s *increaseScore* and *getScoreChange*
         2. Set the goodie to dead and play the appropriate sound
         3. Update Socrates in whichever way the particular goodie affects Socrates
      3. Otherwise, decrease lifetime
      4. If lifetime has expired, set goodie to dead
   2. ***updateSocrates***
      1. Not technically a mutator, as it doesn’t modify the goodie; however, it does modify the gameworld in how it affects Socrates
      2. Declared as pure virtual, since each of the goodies affect Socrates in a different way
3. **Accessors**
   1. ***getScoreChange****:* Returns the change in Socrates’ score there should be when Socrates picks up the goodie
      1. Declared as constant and pure virtual, since each goodie changes Socrates’ score differently
   2. ***actorDamageable***: Returns true, since all goodies are damageable; constant function

# Different Goodies and Fungus Implementations

Each goodie (Flamethrower, *ExtraLife*, and *RestoreHealth*) and fungus (*Fungus*) class is derived from the Goodie base class, and is constructed accordingly. The only two functions declared in each such class are *getScoreChange* and *updateSocrates* functions, which were declared as pure virtual in Goodie.

With *getScoreChange*, each goodie/fungus just returns the change in Socrates’ score on goodie pickup (e.g. 300 for a flamethrower charge goodie).

With *updateSocrates*, each goodie fungus calls appropriate functions in the StudentWorld to affect Socrates per the spec. For example, a RestoreHealth goodie calls StudentWorld’s *restoreSocratesHitPoints* function in its *updateSocrates* implementation, and a Fungus calls StudentWorld’s *decreaseSocratesHitPoints* function to damage Socrates with 20 hit points.

# Bacterium Implementation

The Bacterium class is derived from Actor and serves as a base class (directly or indirectly) for Regular Salmonella, Aggressive Salmonella, and E. Coli bacteria.

1. **Constructor**: *Bacterium(int imageID, double startX, double startY, StudentWorld\* world, int hitPoints)*
   1. Creates an actor with the specified image ID at the specified location, with the specified hit points, 90 direction and 0 depth
   2. Initialises movement plan distance to 0
2. **Pure virtual mutators**:
   1. ***createNewBacterium***: Uses StudentWorld’s template *createBacterium* function to create a new bacterium of the appropriate type
      1. It’s implemented as pure virtual and public, since Regular Salmonella, Aggressive Salmonella and E. Coli derived classes all call *createBacterium* with a different type specification
   2. ***moveBacterium***: Implements movement plan for different kinds of bacteria
      1. Implemented pure virtual since salmonella and E. Coli move differently
3. **Virtual mutators**
   1. ***searchForSocrates***: Checks whether Socrates is within detection radius of the bacterium and acts accordingly; returns a boolean reflecting whether bacterium successfully found and attempted to move toward Socrates
      1. Returns false by default, since only aggressive salmonella search for Socrates (E. Coli’s search algorithm for Socrates works differently, and is implemented in E. Coli’s *moveBacterium* function instead)
      2. Only specialisation of this function is in the derived class Aggressive Salmonella
4. **Non-virtual mutators (i.e. mutators not defined differently in any derived classes)**:
   1. ***doSomething****:* All bacteria follow a similar series of actions, and *doSomething* therefore serves as a common implementation for all of them
      1. Returns immediately if actor isn’t alive
      2. Bacterium searches for Socrates using *searchForSocrates* and stores the return value (false for Regular Salmonella / E. Coli, true/false for Aggressive Salmonella depending on whether the Aggressive Salmonella found and attempted to move toward Socrates)
      3. Checks if bacterium overlaps with Socrates and deals damage as appropriate with StudentWorld’s *decreaseSocratesHitPointsi*
      4. Checks if bacterium has eaten enough food; if yes,
         1. Reproduces using a private member function *reproduce*, which computes new coordinates for the spawned bacterium and uses the *createNewBacterium* public member function to create a new bacterium at the new coordinates of the appropriate type
         2. Resets food eaten to nil
      5. Checks for overlapping food objects and consumes one if any using StudentWorld’s *findAndEatOverlappingFood* function
      6. If Bacteria didn’t find Socrates in step one, executes *moveBacterium* (happens automatically for Regular Salmonella / E. Coli; may or may not happen for Aggressive Salmonella, depending on whether it found and attempted to move toward Socrates already)
   2. ***damageActor***: Since all bacteria take damage similarly, the same implementation works for each of them
      1. Deals specified damage to the bacterium object and plays the appropriate sound
      2. If hit points reach 0
         1. Sets bacteria to dead
         2. Increases player score appropriately with StudentWorld’s *increaseScore* function
         3. With 50% chance, creates a new food object at former bacterium location using StudentWorld’s *createFood* function
   3. ***setActorDead***: Since bacteria also need to inform StudentWorld when they die, the default implementation of setActorDead defined in the Actor class is insufficient for bacteria
      1. Calls Actor’s *setActorDead* to set bacterium to dead
      2. Calls StudentWorld’s *decBacteriaLeft* function to decrement the number of bacteria (both in pits and outside of them) the StudentWorld thinks are left in the game
5. **Non-virtual accessors (i.e. accessors that act the same for all derived bacteria; all constant)**
   1. ***getHitPoints***: Returns the number of hit points of the bacterium
   2. ***getFoodEaten***: Returns the amount of food eaten by the bacterium
   3. ***actorDamageable***: Overrides the default *actorDamageable* function defined in Actor, which returned false; since bacteria are damageable, this returns true instead
6. **Pure virtual accessors (al constant)**: ***getHurtSound****,* ***getDieSound***, and ***getDamage*** are all declared pure virtual since salmonella and E. Coli have different hurt and die sounds, and because all bacteria deal different amounts of damage; these functions are defined public for use in *doSomething* and *damageActor*

# Salmonella Implementation

The Salmonella class is derived from Bacterium and serves as a base class for Regular and Aggressive Salmonella.

1. **Constructor**: *Salmonella(int imageID, double startX, double startY, StudentWorld\* world, int hitpoints)*
   1. Creates a Bacterium object with imageID IID\_SALMONELLA at the appropriate position with the specified hit points
   2. Sets movement plan distance to 0
2. **Mutator *movebacterium*:** Since both Regular and Aggressive Salmonella move similarly (follow movement plan / move towards food / move toward random direction), this implementation works for both regular and aggressive salmonella  
   Note: Aggressive Salmonella have a different movement pattern in that they first search for Socrates; however, this is implemented in the *searchForSocrates* function. *moveBacterium* only describes the movement of movement plan / food-oriented / random movements of salmonella.
   1. If movement plan distance is greater than 0
      1. Decrease movement plan distance
      2. Find position in 3 pixels in the current direction from the current position using GraphObject’s *getPositionInThisDirection*
      3. If new coordinates are valid (i.e. movement wouldn’t be blocked; check via StudentWorld’s *movementValid* function), move to new coordinates
      4. Otherwise, call private member function *resetMovement* (sets direction to random and resets movement plan distance to 10)
   2. Otherwise, search for food
      1. Find direction to closest food object within VIEW\_RADIUS pixels using StudentWorld’s *findClosestFoodDir function*
      2. If food found,
         1. Find coordinates 3 pixels in the food direction from the current position using GraphObject’s *getPositionInThisDirection*
         2. If new coordinates are valid (check via StudentWorld’s *movementValid* function), move to new coordinates and update Direction
         3. Otherwise, call *resetMovement*
      3. Otherwise, call *resetMovement*
3. **Accessors (all const, override accessors declared in Bacterium)**: ***getHurtSound***and ***getDieSound*** return SOUND\_SALMONELLA\_HURT and SOUND\_SALMONELLA\_DIE respectively, for use in ***damageActor***

# RegularSalmonella Implementation

The RegularSalmonella class is derived from Salmonella.

1. **Constructor**: *RegularSalmonella(double startX, double startY, StudentWorld\* world)*
   1. Creates a new Salmonella object with imageID IID\_SALMONELLA and regular salmonella hitpoints
2. There aren’t many new mutators defined in RegularSalmonella; it does, however, define ***createNewBacterium*** (declared pure virtual in Bacterium) to call StudentWorld’s template *createBacterium* function with template type RegularSalmonella
3. Accessor ***getDamage****()* returns regular salmonella damage, for use in *doSomething*

# AggressiveSalmonella Implementation

The RegularSalmonella class is derived from Salmonella.

1. **Constructor**: *AggressiveSalmonella(double startX, double startY, StudentWorld\* world)*
   1. Creates a new Salmonella object with aggressive salmonella hitpoints
2. **Mutator *searchForSocrates*** overrides the virtual *searchForSocrates* function defined in Bacterium (which returned false); it checks whether Socrates is within aggressive salmonella’s detection radius and acts accordingly
   1. Call StudentWorld’s *getSocratesX* and *getSocratesY* functions to check the distance between aggressive salmonella and Socrates
   2. If distance to Socrates <= detection radius (72 pixels),
      1. Using StudentWorld’s *getDirToSocrates* function, find Socrates’ direction from the Aggressive Salmonella’s position
      2. Using GraphObject’s *getPositionInThisDirection*, find the coordinates 3 pixels in Socrates’ direction
      3. If moving to those coordinates is valid (check via StudentWorld’s *movementValid*), move to those coordinates
      4. Return true (Aggressive Salmonella successfully found and tried to move toward Socrates)
   3. Otherwise, return false (couldn’t find Socrates)
3. **Non-mutator** ***createNewBacterium*** (declared pure virtual in Bacterium) calls StudentWorld’s template *createBacterium* function with template type AggressiveSalmonella
4. Accessor ***getDamage****()* returns aggressive salmonella damage, for use in *doSomething*

# EColi Implementation

The EColi class is derived from the Bacterium class.

1. **Constructor**: *EColi(double startX, double startY, StudentWorld\* world)*
   1. Creates a Bacterium with the specified coordinates, imageID IID\_ECOLI, and E. Coli hitpoints
2. **Mutator *moveBacterium***defines the moveBacterium function declared pure virtual in the Bacterium class with E. Coli’s unique movement plan
   1. Find direction to Socrates using StudentWorld’s *getDirToSocrates* function
   2. Repeatedly 10 times:
      1. Find coordinates 2 pixels forward in Socrates’ direction using StudentWorld’s *getPositionInThisDirection* function
      2. If moving to new coordinates is valid (check via StudentWorld’s *movementValid* function,
         1. Set direction to direction to Socrates
         2. Move to new coordinates
         3. Return
      3. Otherwise, increment angle to Socrates by 10 and try again
3. **Non-mutator *createNewBacterium*** (declared pure virtual in Bacterium) calls StudentWorld’s template *createBacterium* function with template type EColi
4. **Accessors (all const, override accessors declared in Bacterium)**: ***getHurtSound***and ***getDieSound*** return SOUND\_ECOLI\_HURT and SOUND\_ECOLI\_DIE respectively for use in *damageActor*, and ***getDamage*** returns E. coli damage

Implementation of StudentWorld

StudentWorld is implemented with several public member functions used by Actors, and a few auxiliary private member functions used by StudentWorld methods. A pointer to Socrates is stored in *m\_socrates*, pointers to all other actors are stored in a set *m\_actor*, and a private data member *m\_bacteriaLeft* tracks the total number of bacteria left/alive in the dish, including the bacteria not yet released by pits.

# Auxiliary private member functions

Implementation not described in detail, since spec only asked for a description of public member functions)

1. ***findValidCoords***: Find valid coordinates for new pits / food objects / dirt piles (i.e. repeatedly generates random coordinates until valid coordinates found; each time, checks for whether new coordinates are within 120 pixels from the dish center and overlap with already created pits / food objects / dirt piles); auxiliary function for *init*
2. ***removeDeadActors***: Iterates over *m\_actors* to delete and remove any dead actors from the game; auxiliary function for *move*
3. ***addNewGoodies***: Simulates the chance of fungus/goodie creation and creates such objects at random locations on the circumference of the dish, per spec; auxiliary function for *move*
4. ***getStatText***: Uses stringstream library to return the status line text of the game; auxiliary function for *move*

# Public member functions

## *init* function

set *m\_socrates* to point to a new Socrates object

set *m\_bacteriaLeft* to level number \* total initial bacteria (i.e. the number of pits times the number of bacteria initially in each pit, since number of pits = level number)

create as many pits as the level number (retrieved by GameWorld’s *getLevel* function)

for each pit to be created, find valid coordinates using *findValidCoords*

create a new pit and insert its pointer into *m\_actors*

create min(5 \* L, 25) food objects (where L = level)

for each pit to be created, find valid coordinates using *findValidCoords*

create a new pit and insert its pointer into *m\_actors*

create a set to temporarily keep pointers to all the new dirt piles (since dirt piles *can* overlap with each other and *findValidCoords* checks for overlap with objects in *m\_actors*, we don’t want to insert dirt piles into *m\_actors* as and when they’re created)

create max(180 - 20 \* L, 20) dirt piles

for each dirt pile to be created, find valid coordinates using *findValidCoords*

create a new dirt pile and insert its pointer into the temporary dirt pile set

iterate over temporary dirt piles set and insert all its pointers into *m\_actors*

return GWSTATUS\_CONTINUE\_GAME

## *move* function

ask Socrates to do something

for each actor pointer in m\_actors:

make actor do something

if Socrates died, calling GameWorld’s *decLives* and return GWSTATUS\_PLAYER\_DIED

if no bacteria left (check by StudentWorld’s *getBacteriaLeft*), return GWSTATUS\_FINISHED\_LEVEL

remove dead actors by calling StudentWorld’s *removeDeadActors*

add new goodies by calling StudentWorld’s *addNewGoodies*

call StudentWorld’s *getStatText* to get status line and update status line using GameWorld’s *setGameStatText*

return GWSTATUS\_CONTINUE\_GAME

## *cleanup* function

if *m\_socrates* isn’t nullptr, delete Socrates and set it to nullptr

call delete on each pointer stored in *m\_actors*

clear *m\_actors* to get rid of dangling pointers

## Destructor

StudentWorld’s destructor just calls *cleanUp*.

## Functions that access or modify *m\_bacteriaLeft*

1. ***getBacteriaLeft***: returns the number of bacteria left in the dish (including bacteria not yet released by pits)
2. ***decBacteriaLeft***: Decrements the number of bacteria left in the dish by 1 (called by Bacterium’s *setActorDead* to notify the StudentWorld)

## Functions that create new objects

1. ***createFood***: Creates a new food object at the specified location and inserts a pointer to it into *m\_actors*
2. ***createProjectile***: template function that creates a new projectile of the specified type at the specified coordinates (and with given direction) and inserts a pointer to it into *m\_actors*
3. ***createBacterium***: template function that creates a new bacterium of the specified type at the specified coordinates and inserts a pointer to it into *m\_actors*

## Functions that modify non-Socrates objects

1. ***findAndDamage*** searches for damageable, alive actors that overlap with the given (x, y) coordinates and damages them with given damage amount

*for each actor in m\_actors:  
 if actor is alive, damageable, and is SPRITE\_WIDTH distance or closer:  
 damage actor  
 return true*

*return false // didn’t find any actor to damage*

1. ***findAndEatFood*** searches for a food object that overlaps with the given (x, y) coordinates and “eats” (i.e. sets to dead) such a food object if found

*for each actor in m\_actors:*

*if actor is eatable, alive, and SPRITE\_WIDTH distance or closer:*

*set actor to dead*

*return true*

*return false // didn’t find any overlapping eatable actor*

## Functions that modify or access Socrates

1. **Mutators**:
   1. ***restoreSocratesHitPoints***: resets Socrates’ hitpoints by calling Socrates’ *restoreHitPoints* function
   2. ***decreaseSocratesHitPoints***: decreases Socrates hitpoints by a specified amount by calling Socrates’ *damageActor*
   3. ***increaseSocratesFlames***: increases Socrates flames by a specified amount by calling Socrates’ *increaseFlames*
2. **Accessors:**
   1. ***getSocratesX, getSocratesY*** return Socrates’ x and y coordinates by calling Socrates’ *getX* and *getY* respectively
   2. ***getDirToSocrates***: returns the direction to Socrates from specified (x, y) by using *atan2* and passing in as parameters the difference in specified (x, y) and Socrates (x, y)

## Functions that retrieve some information about the world and world objects

1. ***movementValid*** checks whether the given (x, y) coordinates can be moved to; in other words, returns whether (x, y) is within arena and doesn’t overlap with an object that blocks movement; used by all bacteria

*if given (x, y) not less than VIEW\_RADIUS distance from dish center, return false*

*for actor in m\_actors:*

*if actor blocks movement and distance between actor and given (x, y) <= SPRITE\_WIDTH / 2*

*return false*

*return true // coordinates in dish and no overlapping blocking object found*

1. ***findClosestFoodDir*** returns whether there is any food object within VIEW\_RADIUS distance of given (x, y), and modifies a Direction parameter to the direction to that food object from given (x, y); used by Salmonella objects

*set a boolean to track whether food has been found (say, foodFound) to false*

*for each actor in actors:*

*if actor is eatable:*

*if distance to actor is closest found so far (and within VIEW\_RADIUS pixels):*

*set foodFound to true*

*update closest food object direction found so far (computed by using atan2)*

*update closest food object distance found so far*

*if foodFound: update the direction parameter passed into function to closest food object direction*

*return foodFound*

Assumptions

1. The specification says to generate new goodies and fungus objects at a random position on the circumference of the disk *exactly* VIEW\_RADIUS (=128) pixels from the center of the dish; however, since the distance of any such random location is a square root (Pythagorean theorem), it will be close to but not exactly 128 pixels. Since it’s impossibly to generate random locations such that they’re always exactly 128 pixels from the center, I assumed it’s fine if the locations are approximately 128 pixels.
2. The specification mentions that E. Coli start with a movement plan distance of 0; however, since E. Coli objects never really care about their movement plan distance and always just move toward Socrates, I didn’t give E. Coli objects any functionality to do with movement plan distances. In other words, the E. Coli class doesn’t have a private data member tracking its movement plan distance or accessors/mutators that work with its movement plan distance; only the Salmonella base class does so, since only Regular/Aggressive Salmonella care about movement plans. I figured this would improve (if only marginally) the size of the executable and the memory efficiency of the program, and assumed (for the reasons stated above) it wouldn’t change the game’s functioning in any way.
3. On page 38 of the spec, it says “If food can be found within 128 pixels, the regular salmonella will try to move toward that food.” It was unclear as to whether a salmonella bacterium (this applies to both aggressive and regular salmonella) is expected to move toward food in the same tick; in my implementation, I assumed it should.