Ethan Wong

CS32

Smallberg

Project 3 Report

Description of Functions in Classes:

Base Actor Class

Actor(int ImageID, double startX, double startY, int startDirection, int depth, StudentWorld\* petriDish, double HP, bool damageable);

* Constructor for an Actor Object, initializes member variables accordingly

virtual ~Actor() ;

* Virtual destructor for the actor class

bool deadOrAlive();

* Returns whether or not a actor is alive

virtual void doSomething() = 0;

* Tells an actor to do something (pure virtual because each derived class will have their own definition)

void setDead();

* Mark an actor as dead

StudentWorld\* getWorld();

* Get the StudentWorld that the actor resides in

void setObjectType(std::string s);

* Used to classify what type an actor is (Socrates, Salmonella, etc)

std::string returnObjectType();

* Returns what type an actor is

bool isItDamageable() const;

* Checks whether or not an actor can be damaged

void takeDamage(int damage);

* Decrements an actor’s health points

double getHP();

* Returns the actor’s health point value

void setHP(double hp);

* Sets the actor’s health point value (cannot exceed 100 health points)

virtual bool isEdible();

* Checks if this actor is a food object (virtual because I will redefine this in Food class)

virtual bool isDirty();

* Checks if this actor is a dirt object (virtual because I will redefine this in Dirt class)

double euclideanDistance(const Actor\* actor1);

* Finds the distance between two separate actors

virtual bool isBacteria();

* Returns true if the actor is a derived from the bacteria class

virtual bool isGoodie();

* Returns true if the actor is derived from the goodies class

Socrates Class (derived from Actor class)

Socrates(StudentWorld\* petriDish);

* Constructor for the Socrates class, initializes member variables accordingly

virtual ~Socrates();

* Virtual destructor for the Socrates class

virtual void doSomething();

* If Socrates is alive:
* If the user hits left, move Socrates 5 degrees counterclockwise around the circle and adjust his direction variables accordingly
* If the user hits right, move Socrates 5 degrees clockwise around the circle and adjust his direction variables accordingly
* If the user hits enter, shoot flames in a circle surrounding Socrates
* If the user hits space, shoot a spray in the direction that Socrates is facing
* If the user doesn’t hit a key, increase the spray ammunition by one

virtual void addHP(int amount);

* Add health points (can use negative integer to decrease health points, can’t exceed 100 health points)

int getSprayAmount();

* Returns how many sprays Socrates has left

void shootSpray();

* Shoots a spray in the direction Socrates is facing; decrements spray count by one

int getFlameAmount();

* Returns how many flames Socrates has left to shoot

void shootFlame();

* Shoots sprays in a circle around Socrates; decrements flame amount by one

void increaseFlameAmmo();

* Increases the amount of flames Socrates has by one

Projectile Class (derived from Actor)

Projectile(int ImageID, double startX, double startY, int direction, int depth, StudentWorld\* petriDish);

* Constructor for the projectile class

virtual ~Projectile();

* Virtual destructor for the projectile class

virtual void doSomething() = 0;

* Tells the projectile to do something (pure virtual because the derived classes will redefine it)

virtual bool distanceTraveled(double a, double b) = 0;

* Checks if distance the projectile has traveled exceeds a certain amount (pure virtual because derived classes will redefine it)

Flame Class (derived from Projectile)

Flame(double startX, double startY, int direction, StudentWorld\* petriDish);

* Constructor for the flame class, initializes member variables accordingly

virtual ~Flame();

* Virtual destructor for the flame class

virtual void doSomething();

* If the flame isn’t dead, it go out and destroy dirt piles, goodies, fungi, and bacteria that are in its path
* If the flame has traveled its maximum distance it will mark itself as dead

virtual bool distanceTraveled(double a, double b);

* Returns false if the flame has traveled 32 or more pixels; returns true otherwise

Spray Class (derived from Projectile)

Spray(double startX, double startY, int direction, StudentWorld\* petriDish);

* Constructor for the spray class, initializes member variables accordingly

virtual ~Spray();

* Virtual destructor for the spray class

virtual void doSomething();

* If the spray isn’t dead, it go out and destroy dirt piles, goodies, fungi, and bacteria that are in its path
* If the spray has traveled its maximum distance it will mark itself as dead

virtual bool distanceTraveled(double a, double b);

* Returns false if the spray has traveled 112 or more pixels; returns true otherwise

Goodie Class (derived from Actor)

Goodies(int ImageID, double angle, StudentWorld\* petriDish);

* Constructor for the goodie class, initializes member variables accordingly

virtual ~Goodies();

* Virtual destructor for the goodie class

virtual void function() = 0;

* Defines what each goodie will do when Socrates overlaps with them (pure virtual because it will be redefined in each derived class)

virtual void doSomething();

* Sets the goodies as dead if they have lived passed their allotted lifetime
* If the goodie is alive and Socrates collides with one, do its assigned function

Fungi Class (derived from Goodie Class)

Fungi(StudentWorld\* petriDish);

* Constructor for the fungi class

virtual ~Fungi();

* Virtual destructor for the fungi class

virtual void function();

* If Socrates comes in contact with the fungi, he loses 20 health points
* Decreases score by 50

Restore Health Goodie Class (derived from Goodie Class)

HealthPack(StudentWorld\* petriDish);

* Constructor for the health pack class

virtual ~HealthPack();

* Virtual destructor for the health pack class

virtual void function();

* If Socrates comes in contact with the health pack, his health points are restored
* Increases score by 250

Extra Life Goodie Class (derived from Goodie Class)

ExtraLifePack(StudentWorld\* petriDish);

* Constructor for the extra life pack class

virtual ~ExtraLifePack();

* Virtual destructor for the extra life pack class

virtual void function();

* If Socrates comes in contact with the extra life goodie, he gains an extra life
* Increases score by 500

Extra Flame Goodie Class (derived from Goodie Class)

FlamePack(StudentWorld\* petriDish);

* Constructor for the extra flame pack class

virtual ~FlamePack();

* Virtual destructor for the flame pack class

virtual void function();

* If Socrates comes in contact with the Extra Flame Pack goodie, he gains five flames
* Increases score by 300

Dirt Class (derived from Actor Class)

Dirt(double startX, double startY, StudentWorld\* petriDish);

* Constructs dirt objects at a specified location

virtual void doSomething();

* Tells dirt to do something (It just sits there)

virtual ~Dirt();

* Virtual destructor for dirt

virtual bool isDirty();

* Returns true to indicate that this object is a dirt object

Food Class (derived from Actor Class)

Food(double startX, double startY, StudentWorld\* petriDish);

* Constructs food objects at a specified location

virtual void doSomething();

* Tells pizza to do something (it just sits there)

virtual ~Food();

* Virtual destructor for the dirt

bool isEdible();

* Returns true to indicate that this object is a food object

Pit Class (derived from Actor class)

Pit(double startX, double startY, StudentWorld\* petriDish);

* Constructs pit at a specified location

virtual void doSomething();

* If the pit is empty it will set itself as dead
* Tells pit to do something
* If the pit isn’t dead and still has bacteria left inside of it, it will randomly emit one of the bacteria stored inside it

virtual ~Pit();

* Virtual destructor for the pit

bool isDepleted();

* Returns whether or not the pit has emptied all of its bacteria into the world yet

int returnSalmonella();

* Returns the amount of salmonella that are still inside the pit

int returnAggressive();

* Returns the amount of aggressive salmonella that are still inside the pit

int returnEcoli();

* Returns the mount of E Coli that are still inside the pit

Bacteria Class (derived from Actor Class)

Bacteria(int ImageID, double startX, double startY, int hp, StudentWorld\* petriDish);

* Constructs bacteria at a specified location with specified health point value

virtual ~Bacteria();

* Virtual destructor for bacteria

virtual void doSomething() = 0;

* Tells the bacteria to do something (virtual void because its derived classes will define it)

void eatFood();

* Bacteria will eat a food, causing the food to disappear
* Will increase the bacteria’s count of how many foods it has eaten by one

int returnFoodEaten();

* Returns the number of foods that the bacteria has eaten

int returnMvmtPlanDist();

* Returns the amount of pixels the bacteria is planning to move in a certain direction

void setMvmtPlan(int x);

* Gives the bacteria a value (in pixels) to indicate how many pixels it plans to move in that direction

void setFoodEaten(int amount);

* Sets the amount of food the bacteria has eaten

bool isBacteria();

* Returns true to indicate that this actor is a bacteria

Salmonella Class (derived from bacteria class)

Salmonella(double startX, double startY, StudentWorld\* petriDish);

* Constructs a new salmonella at a specified location; sets health point value to 4

virtual ~Salmonella();

* Virtual destructor for salmonella objects

virtual void doSomething();

* If the salmonella is dead, then return
* If the salmonella hits Socrates, reduce his health points by one
* If the salmonella has eaten 3 foods, it will spawn another salmonella
* If the salmonella overlaps with food, the food will die and the count of how many foods the salmonella has eaten will increase by one
* If the salmonella has a plan to move, it will move in that direction and decrements its movement plan distance
* If the salmonella has a plan to move but its movement is blocked by a dirt or the edge of the circle, it will choose a random direction and try to move that way instead
* If the salmonella has no plan to move, it will see if there are any food objects within 128 pixels of it and attempt to move towards the food
* If there is movement blockage once again, the salmonella will randomize its direction again and try to move that way and return

Aggressive Salmonella Class (derived from bacteria class)

aggressiveSalmonella(double startX, double startY, StudentWorld\* petriDish);

* Constructs a new salmonella at a specified location; sets health point value to 10

virtual ~aggressiveSalmonella();

* Virtual destructor for salmonella objects

virtual void doSomething();

* If the aggressive salmonella is dead, then return
* If the aggressive salmonella is within 72 pixels of Socrates, it will try to move towards Socrates (or get stuck on a pile of dirt if it runs into a dirt pile)
* If the aggressive salmonella hits Socrates, reduce his health points by two
* If the aggressive salmonella has eaten 3 foods, it will spawn another aggressive salmonella
* If the aggressive salmonella overlaps with food, the food will die and the count of how many foods the aggressive salmonella has eaten will increase by one
* If the aggressive salmonella has a plan to move, it will move in that direction and decrement its movement plan distance
* If the aggressive salmonella has a plan to move but its movement is blocked by a dirt or the edge of the circle, it will choose a random direction and try to move that way instead
* If the aggressive salmonella has no plan to move, it will see if there are any food objects nearby and attempt to move towards the food
* If there is movement blockage once again, the aggressive salmonella will randomize its direction again and try to move that way and return

Ecoli Class (derived from Bacteria)

Ecoli(double startX, double startY, StudentWorld\* petriDish);

* Constructs a new E coli at a specified location, sets its health point value to 5

virtual ~Ecoli();

* Virtual destructor for E coli objects

virtual void doSomething();

* If the E coli is dead, then return
* If the E coli overlaps with Socrates, it will deal five damage
* If the E coli has eaten 3 foods, it will spawn another E coli
* If the E coli overlaps with food, the food will die and the count of how many foods the E coli has eaten will increase by one
* The E Coli will repeat the following process up to ten times:
* If the E coli is within 256 pixels of Socrates, it will get the direction towards Socrates and try to move towards Socrates. If the E coli is blocked by dirt, it will increase its direction by ten degrees and try to move towards Socrates again
* If the process repeats ten times and it never works, the E coli will just stay still

StudentWorld Class

StudentWorld(std::string assetPath);

* Constructor for the StudentWorld class

virtual ~StudentWorld();

* Virtual destructor for the StudentWorld class

virtual int init();

* Sets the initial state for the game – adds in Socrates player, adds in dirt, foods, and pit(s)
* declared as virtual just for clarity since this is pure virtual in the parent class

virtual int move();

* Allows the game to build, run, and terminate
* Checks for collisions between actors, allows actors to do something, removes dead actors from the game, indicates when the level is finished, displays the header at the top of the screen
* declared as virtual just for clarity since this is pure virtual in the parent class

virtual void cleanUp();

* Frees all the actors in the game; destroys the level
* declared as virtual just for clarity since this is pure virtual in the parent class

void accessMapActors(string actor);

* decrements the amount of actors map that keeps track of how many of each actor are present in the game

const vector <Actor\*> & returnVec\_Actors();

* returns the vector of actors

Socrates\* returnSocrates();

* returns Socrates pointer

double findDistance(double x1, double y1, double x2, double y2);

* used to find the distance between two actors

bool overlap(double x, double y);

* checks if two actors overlap

bool sprayDamage(Actor\* a1);

* deals with how the sprays damage and interact with other actors when they come in contact

bool flameDamage(Actor\* a2);

* - deals with how the flames damage and interact with other actors when they come in contact

void addGoodies();

* Method for adding the goodies randomly into the game
* Responsible for the goodies’ random location; random lifetime; goodie despawn

void addDirt();

* Adds dirt randomly into the game within the circle (not on the perimeter of the circle though)

void addPit();

* Adds pit(s) randomly into the game within the circle (not on the perimeter of the circle though)

void addFood();

* Adds pit(s) randomly into the game within the circle (not on the perimeter of the circle though)

bool otherCollision(Actor\* a);

* Returns whether or not two actors have collided with each other

void hurtSocrates(int damage);

* Decreases Socrates health points by a specified value

void addActor(Actor\* a);

* Adds an actor into the game

bool foodOverlap(Actor\* a);

* Returns whether or not an actor has overlapped with a food object

bool movementBlockage(Actor\* a);

* Returns whether or not a bacteria’s movement is blocked by a dirt pile or by the edge of the circle

bool isFoodNearby(Actor\* a);

* Returns whether or not there is food within 128 pixels of an actor(bacteria), sets the bacteria’s direction to head towards that food

bool closeToSocrates(Actor\* a);

* Returns whether or not an actor(aggressive salmonella) is within 72 pixels of Socrates, sets that actor’s direction to head towards Socrates

bool isEcoliCloseToSocrates(Actor\* a);

* Returns whether or not an actor(E Coli) is within 256 pixels of Socrates, sets that actor’s direction to head towards Socrates

double randomDecimal();

* returns a random decimal between greater than 0 and less than 1

bool attachedToSocrates(Actor\* a);

* this returns true if an aggressive salmonella has latched onto Socrates

int returnMapActors(string n);

* this returns how many of a certain type of actor there are currently in the game

bool wonLevel();

* this returns true when the player has eliminated all the bacteria on the screen and there are no more pits remaining

Description of bugs/unfinished parts

There aren’t any parts of the project that I failed to implement in this program. I also don’t have any known bugs in my program. I don’t think this is a bug, but my game runs a bit slower than the game provided on the website. I don’t believe that this was a result of my code being faulty, but instead because of my computer being very old. It may be a result of my code causing some lag with collision detection but I believe that the main cause of the lag is stemming from my computer being very old and slow (my laptop is from 2012 and doesn’t even have the capacity to play any video games, so I think my computer just has very low processing power). Other than that, my games works as specified. I apologize if my game is still a bit slow on another computer; no matter what I did I could not seem to get the game to run full speed as it was supposed to.

List of assumptions I made about the project

In this project I assumed that minor lag would not be viewed as a major problem. I tried to look for ways to reduce the lag, but I could not find an implementation that succeeded in doing so. For instance, I tried to implement a function for collision detection that would not check objects that did not need to be checked. I tried to have an implementation where dirt would not check if it overlapped with another dirt, because this is legal. I implemented it correctly but it still did not make much difference in the lag (again, probably because my computer is very slow). I eventually went to TA office hours and was told by the TA there at the time that it would not be a problem that my game ran slightly slower.

In addition to this, I assumed that the spread of the flames would not be exactly 22 degrees between each flame sprite. This is because if there are 16 frames shot out at a time with one key press, 16 \* 22 = 352. This means that the flames are spread 352 degrees around Socrates, instead of a complete circle. There will be a slightly larger gap between two of the flames. Other than these two assumptions, I found that the specification for the project was sufficiently clear. I asked the TAs, LAs, and Professor Nachenberg for any other misunderstandings I had about the specification, which helped to clean up any issues I was running into. Everyone I asked for help gave me clear answers.

Testing the Classes

Actor Class

I didn’t really have any direct testing for this class because it was the base class that I derived all my other classes from. The main testing I did for this class was to use print statements (cout statements) in order to see if the program was properly accessing the functions. For instance, if I wanted to test whether the getHP() function was working properly, I would add a cout statement inside the function and see if it was being outputted in the terminal.

Socrates Class

The main way that I tested this class was to play the game and see if Socrates was doing his specified behaviors. For instance, I would test if the key presses were doing the correct functions: space would shoot a spray, enter would shoot flames, etc. Once again I used print statements like I did to test the Actor class. I would also test whether Socrates was getting damaged properly and if his ammunition amounts were adjusting properly according to what key presses I was clicking. Once again, most of the testing was done by playing the game and comparing it with the game posted on the website.

Projectile Class (Flames and Sprays)

I tested the two different types of projectiles similarly to the way I tested the Socrates class. I played the game and made sure that when the projectiles collided with other objects, they were being properly damaged and/or destroyed. For instance, I made sure that the projectiles were destroying the dirt and goodies on impact. I also made sure that the projectiles weren’t damaging actors that they weren’t supposed to be damaging (such as the pit or the food actors). I was also struggling with the hit detection part of the projectiles so I used plenty of cout statements and the debugger to trace through the code and see where my code was breaking or doing undesired behavior. I had an issue where my sprays were disappearing even though they didn’t come into contact with anything. I later realized it was because my hit detection function was not logically correct. I used incorrect math in my implementation. It took me quite a long time to debug the projectiles, but the practice of using the debugger and commenting out code I wasn’t sure about eventually led me to completing the implementation of the projectiles.

Goodie Class (Fungi, Extra Life Pack, Restore Health Pack, Flame Charge Pack)

Since all the goodies behave quite similarly to each other, I tested and debugged them all the same way. I made sure that the goodies were disappearing upon impact and affecting Socrates and the score as they were coming in contact with Socrates. I had an issue with my Extra Life Pack because it wasn’t actually incrementing the amount of lives the player had. I debugged this by putting a cout statement in my increaseLives() function. I realized that it wasn’t actually being called and then later discovered that an incLives() function had already been added in the skeleton code. I used this instead and my goodies functionality worked properly. I also played the game to make sure that the goodies were spawning randomly on the perimeter of the circle and at random time intervals. I also observed them and checked to see if they were despawning after an appropriate amount of time. Thankfully I didn’t have too many issues with the goodie classes as I was able to get most of the goodies’ functionality correctly on the first try.

Dirt Class

The dirt class was the first class I implemented. I didn’t really have any errors or bugs with the creation of the dirt, but I was running into an issue where my dirt were clumping up in the middle of the circle. I didn’t pay much attention to it until my TA mentioned that this was a result of using polar coordinates. I then implemented a function that created a random decimal between 0 and 1 and multiplied it by a random angle “theta” and a random radius. This ensured that the dirt was spawning truly randomly within the circle of radius 128 pixels inside the game’s circle. I also made sure to test whether the dirt were being destroyed properly and spawning randomly by opening and closing the game multiples times and checking their distribution based on how I added them in StudentWorld’s init() function.

Food Class

My food class was quite problematic. I was unable to find a way to prevent the food from overlapping with each other or with other actors in the game upon spawning. I eventually remedied this by creating a function called overlap() that would check if the food’s potential spawn location would be overlapping with anything else. If it was overlapping, it was to create rerandomize another spawning location until it could find a suitable place to spawn. I tested this functionality out by repeatedly opening and closing the game to make sure that the food never overlapped with anything else after the initial startup of the game. Implementing the overlap function was quite difficult and required the use of a lot of print statements and math to see if my logic was correct. I had a bug that stemmed from me forgetting a parenthesis that I was able to catch using the debugger as well. I also played the game to make sure that the food was disappearing properly when a bacteria came in contact with it.

Pit Class

For the spawning of the pit class, I used similar logic to what I did with the Food class. I used the overlap() function (after I finished debugging it) to make sure that the pit spawned in a location where it wasn’t overlapping with any other actors. For the pit’s doSomething() function, I had bugs where my E coli were not spawning properly. I was using randInt() to ensure that the pit’s choice of what was spawning was truly random and selecting from what was remaining inside of the pit. I traced through the code on a piece of paper and discovered that my randInt() was incorrect because it would have been statistically impossible to spawn an E coli. As I traced out the code on a piece of paper I was easily able to find my bug and adjusted the doSomething() class accordingly so that the correct bacteria were spawning and that the pit would despawn after releasing all of its bacteria. After testing out this logic I played the game and paid close attention to the bacteria that the pit was spawning as well as checking to make sure it wasn’t being damaged by anything.

Bacteria Class (Salmonella, Aggressive Salmonella, E coli)

As I expected, the bacteria were the hardest portion of this project to implement. I developed incrementally, making sure that my salmonella worked properly before moving on to the aggressive salmonella and then the E coli. My main way of testing for the bacteria was to play the game and watch how they behaved. I checked how they were moving, whether they were eating food, whether they were dividing after eating three food, etc. I just compared the behavior of the bacteria to the specification and the official game and made sure they followed the same bevaior. My salmonella had an issue where it kept spinning in circles randomly. I used the debugger and found that it was getting stuck in a loop of changing its own direction because it believed that it was colliding into a pile of dirt. I was able to fix this after using the debugger along with cout statements to see what function call was causing my salmonella to repeatedly change direction. After making sure that my normal salmonella worked properly, I moved onto the aggressive Salmonella. Once again, my bacteria was spinning around randomly once it came in contact with Socrates. I found that it was a similar issue with the regular Salmonella. It was because once the aggressive Salmonella latched onto a stationary Socrates, it would detect that it was too close to the edge of the game’s outer circle and attempt to change direction. To resolve this issue, I made a Boolean function that would return true if the aggressive Salmonella was within 8 pixels of Socrates. If the Boolean returned true, then I would prevent the aggressive Salmonella from changing directions (while it was latched on to a stationary Socrates). This prevented the aggressive Salmonella from needlessly spinning in circles. Finally, I implemented my E coli class. As I had learned from my mistakes in implementing the previous two bacteria classes, I thankfully did not have any errors with implementing my E coli class. Thankfully, I was able to implement this class without any bugs.

StudentWorld Class

This class had a lot of functions that I would use to run the game. I had quite a few bugs spread throughout this class’s functions. The main way I debugged this function was to set breakpoints and use the debugger to find out where things were going wrong. I also used many cout statements to check whether or not the functions were being called properly when I wanted them to be called. This, combined with just playing the game and comparing my game’s behavior with the official game, was quite enough for testing. For instance, I had an issue where my game was unable to start. I used the debugger to find out that the program was running through the same loop infinitely because I forgot to increment my iterator. Without the debugger and plenty of cout statements, I would have been very confused on what I needed to fix and what logic errors I was making. The main issue I had with this class was the problem of my game not starting. Every other bug was quite small and I was able to resolve it by using the debugger and scrutinizing each code line by line to see where I had flaws in my logic.