Rodger Lee

Com Sci 32

Professor Smallberg

UID: 204-914-719

Project 3: Kontagion

1.

Actor Class

virtual void doSomething() = 0;

* This function allows each Actor to do an individual action during each tick of the game. I chose to define a pure virtual version of the doSomething() function in my base Actor class because all Actors “do something” in a different way.

bool isAlive() const;

* This function tells whether an Actor is alive or not. It’s in the Actor class because all Actors have an alive/dead state. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden.

void die();

* This function sets an Actor’s state to dead. It’s in the Actor class because all Actors can get killed somehow. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden: just set state to dead.

StudentWorld\* getWorld() const;

* This function returns the StudentWorld that an Actor resides in, so the world can then do operations on other objects in the world. It’s in the Actor class because all Actors have a StudentWorld. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden: just return the Actor’s world.

virtual bool isedible() const;

* This function returns whether or not an Actor is an edible object, which is useful in seeing how objects should behave after experiencing collision. Edible objects usually have an effect on the Actor colliding with it, and it immediately dies after collision. Edible objects include goodies, fungus, and pizza.

virtual bool isenemy() const;

* This function returns whether or not an Actor is an enemy to Socrates. Bacterium, food, and pits are enemies. Enemies that are edible cannot be hit by projectiles, but they can also be consumed by bacteria. Enemies that are not edible can be hit by projectiles.

virtual bool isprojectile() const;

* This function returns whether or not an Actor is a Projectile, which is useful in determing of something collided with a Projectile. It’s in the Actor class because Projectile is a type of Actor, so each Actor is or is not a Projectile. I made it virtual since by default most Actors are not Projectiles, but in the Projectile class, it needs to be overridden to return true.

virtual bool isdamageable();

* Isdamageable objects can be hit by projectiles. DamageableActors have virtual bool isdamageable declared as true, and its default is false. Dirts, goodies, and bacteria are damageable objects.

virtual bool isdirt ();

* Isdirt differentiates dirt objects with all other objects. Isdirt helps studentworld::init() find objects that are overlapping with food and pits. Dirts can overlap each other during init. Additionally, dirts do not have health but die immediately when hit by a projectile.

virtual void losehp(double amt);

* Base class Actor’s losehp returns automatically, since only derived classes use losehp with specialized functions. DamageableActors reduce HP with their void losehp and salmonella and ecoli play a special sound each time it takes damage and dies.

DamageableActor Class

double hp() const;

* This function returns the number of hit points a DamageableActor has. It’s in the DamageableActor class since all DamageableActors have hit points, but not all Actors have hit points. It’s non-virtual since it works the same for all DamageableActors: just return the number of hit points.

void gainhp(double amt);

* This function increases the number of hit points a DamageableActor has by amt. It’s in the DamageableActor class since multiple types DamageableActors can possibly increase their hit points. It’s non-virtual since it works the same for all DamageableActors: just increase the number of hit points.

virtual void losehp(double amt);

* This function decreases the number of hit points a DamageableActor has by amt. It also has the user indicate the cause. It’s in the DamageableActor class since multiple types DamageableActors can possibly decrease their hit points.

Dirt Class

virtual void doSomething();

* This function does not do anything.

Socrates Class

virtual void doSomething();

* This function checks if the Socrates is alive, responds to any feedback from the player (move or shoot projectile), and spray/flame energy points. It’s in the Socrates class because the Socrates executes its own version of doSomething. It’s virtual since it is overriding the virtual Actor doSomething.

void increaseflamechargers();

* This function increases the Socrates’ flame charges by 1. It’s in the Socrates class because only the Socrates can increase its flame charges. It’s non-virtual since it does not need to be overridden.

int flamecharges() const;

* This function returns the Socrates’s amount of flame charge points. It’s in the Socrates class because only the Socrates has flame charge points. It’s non-virtual since it does not need to be overridden.

int spraycharges() const;

* This function returns the Socrates’s amount of spray points. It’s in the Socrates class because only the Socrates has spray points. It’s non-virtual since it does not need to be overridden.

Food Class

virtual void doSomething();

* Food does nothing

Pit Class

Virtual void doSomething();

* Checks how many bacterium are left in the pit and tracks the number to see when it should go away. Once all bacterium are emitted, the pit dies.

Projectile Class

virtual bool isProjectile() const;

* This function returns true (that the Projectile object is indeed a Projectile). It’s in the Projectile class so the StudentWorld can see if an object is a Projectile. It’s virtual since it’s overriding the Actor version of the function.

Goodie Class

virtual void doSomething();

* Goodies check if it collides with Socrates and apply effects and increase points when collided. Its virtual because doSomething is called for all actors in the studentworld file.

Virtual bool isexpired()

* Lets the user know when a goodie has reached its expiration, and it dies at that moment. This is nonvirtual since only goodies have this feature.

Enemy Class

Virtual bool isenemy

Virtual int movementplandistance();

* this function tells us how many more pixels are left in the bacterias movement path. M\_movementplandistance is a private member variable of enemy.

Virtual void moveandcheckisblocked();

* return true If object is blocked, and moves the object forward if object is not blocked. An object is blocked when it collides with a dirt or when it moves out of bounds.

Virtual void checkcollisionwithfood();

* checks if enemy collides with food, and if it does, increase foodeatencount. This uses the iswithineuclidiean distance function in studentworld, which maintains a distance of sprite radius \* 2.

Virtual void findfood()

* this function is meant for enemies like salmonella that track down food when they reach a certain radius away from that food object. It uses setDirection and foodisnearby to locate the direction the food is in. atan2 is a math function used to locate that specific angle

virtual void checkeatenfoods(double& x, double& y)

* checks eaten food count and if count is greater or equal to 3, generate a new enemy of that same type at a location near its current location. Double x and y are passed by reference because this function will return the new x and y locations of the new enemies.

RegSalmonella, AggSalmonella, EColi Class

RegSalmonella serves as the base class for AggSalmonella and EColi, because they all share very similar doSomething functions as well as losehp functions. AggSalmonella and RegSalmonella are both Salmonella, so they share the same hurt and die sounds. EColi and RegSalmonella have almost identical function calls when it comes to do something.

2. My game has no errors or bugs.

3.I made an assumption about creating the flames, I froze the frame on the demo and counted how many flames were generated. 16 were generated relatively spaced apart, so I ran a for loop and made sure each flame was generated before continuing the program.

4. I tested the Socrates class by checking all of its user response switch cases. Turning left , right, pressing enter and space bar were all very responsive and did what they needed to do.

I had a lot of issue with the Salmonella classes, mainly in their ability to find food. I realized that their find food function was taking too many turns and I was making its functionality too complicated by running Boolean functions through 3 different cases. Additionally, I moved my bacteria without checking its collision which caused a lot of issues with it leaving the circle.

I began to check the expected locations prior to moving my objects, which proved to be successful; there are no bugs.

An interesting issue I had with the bacteria movement occurred when ecoli and aggressive salmonella were chasing Socrates along the edge of the circle. Their movements would often be blocked because they were angled towards the outside of the circle. It turns out that the program was considering their movements to be outside of the allowable range due to a math error, which declared their movements blocked. I was using declaring int variables for their x and y coordinates instead of doubles, which caused a rounding error. This small rounding error turned out to be a pretty tricky bug to fix.

I tested out each class by constantly running the demo game, checking if their behaviors responded correctly. Additionally, I had to declare several cout lines to check if the program would run into unexpected edge cases and enter my Boolean cases.