Dean Jones

005-299-127

**Project 3**

1. StudentWorld

* StudentWorld(std::string assetPath): creates StudentWorld by initializing base GameWorld object with assetPath, setting Socrates pointer to nullptr, and both pits and bacterias to initially 0
* ~StudentWorld(): calls cleanUp to free dynamically allocated memory
* virtual int init(): returns GWSTATUS\_CONTINUE\_GAME after setting up Actors for the level. Initializes Socrates at 180 position in dish and adds L Pits, min(5 \* L, 25) Foods, and max(180 - 20 \* L, 20) Dirt piles so that nothing overlaps except Dirt piles potentially with other Dirt piles.
* virtual int move(): calls doSomething for Socrates and every alive actor. If something causes Socrates to die, GWSTATUS\_PLAYER\_DIED is returned. If the level was finished (no more pits or bacteria), GWSTATUS\_FINISHED\_LEVEL is returned. All dead Actors are removed from the game, and Fungi and Goodies alike are potentially added with a 1 in randInt(0, max(510 - L \* 10, 200) - 1) chance. The game status is updated and displayed at the top of the screen, and if the game should go on, GWSTATUS\_CONTINUE\_GAME is returned.
* virtual void cleanUp(): deletes objects pointed to by Socrates pointer and all other Actor pointers. Sets Socrates pointer to nullptr and empties list of Actor pointers.
* Socrates\* getSocrates() const: returns Socrates pointer
* void addActor(Actor\* a): adds Actor\* a to list
* void decPits(): decreases Pit count by 1 when Pit dies
* void incBacteria(): increases Bacteria count when Pit releases Bacteria
* void decBacteria(): decreases Bacteria count when Bacteria is killed
* Actor\* overlapFood(double x, double y): if a Food overlaps with the input coordinates, its pointer is returned. Else, nullptr is returned.
* Actor\* closestFood(double x, double y, double& dist): returns closest Food pointer with 128 pixels and sets dist to its distance from the coordinates. If no Food is within that range, nullptr is returned.
* Actor\* overlapTarget(double x, double y): returns Actor pointer for anything that can be damaged by a projectile at the coordinates. If nothing qualifies, nullptr is returned.
* bool overlapDirt(double x, double y): if Dirt overlaps with the coordinates, returns true. Else, returns false.

The StudentWorld class was tested through the actors that were constructed for the game. decPits() was indeed called when a Pit was out of bacteria, while incBacteria() and decBacteria() were also called at their right respective times. This was verified by using the game stat text display to show the Pit and Bacteria count for each tick. overlapFood, closestFood, and overlapDirt were tested for all Bacteria; initializing a Salmonella close to or on the food and seeing if the right pointer was returned for example, or putting a Dirt pile between an Ecoli and Socrates. overlapTarget was needed for the correct functioning of Projectiles; it was confirmed that Sprays and Flames indeed damaged Dirt, Goodies, Fungi, and Bacteria but nothing else. The numbers of different objects were printed out for init() to ensure that the proper amounts were created. For move(), specific Actors were initialized as dead and were removed by the function as intended. In addition, when Socrates was set as dead after a specified time the game did indeed end, just as when he was killed by Bacteria. Similarly, when Pits and Bacteria were all extinguished from the level, the correct functionality to indicate the level was finished was tested. getSocrates and addActor were both trivial.

1. Actor

* Actor(StudentWorld\* sw, int imageID, double startX, double startY, Direction dir, int depth, double size, bool target, bool edible, bool blocks): Actor is derived from the GraphObject class (to visually render the Actor) and thus needs all the attributes of coordinates, direction, depth, size, etc. The additional parameter of a StudentWorld pointer was necessary for communication between an Actor and other Actors in the StudentWorld as well as the StudentWorld object itself.
* virtual ~Actor(): included to ensure derived classes properly destruct this (function body does nothing)
* virtual void doSomething() = 0: pure virtual, because no Actor object will ever be made and each Actor type will have a different implementation of this (called for each tick)
* StudentWorld\* getWorld() const: returns pointer to StudentWorld that Actors are in
* bool getAlive() const: returns whether Actor is alive
* void setDead(): sets Actor to dead
* virtual void damage(int hp): calls setDead() by default, as unknown whether Actor will have hp making the input parameter meaningful
* bool isTarget() const: tells Spray/Flame whether Actor should be damaged (is Dirt, Fungus, or Goodie)
* bool isEdible() const: tells Bacteria whether Actor is Food and can be eaten
* bool blocks() const: tells Bacteria whether Actor is Dirt and cannot be crossed

The class could not be tested by making an Actor object since it is abstract, but each public member function was thoroughly tested through calls from derived classes. isTarget(), isEdible(), and blocks() were particularly important for determining what to do in certain overlap situations. getWorld() was consistently used to update the StudentWorld object about the game status as needed. getAlive() and setDead() helped when Actors should be removed, like eaten Food or destroyed Dirt.

1. Socrates: public Actor

* Socrates(StudentWorld\* sw): The constructor only needs to take in a StudentWorld pointer, since the player starts at the same position, orientation, and has all the other image attributes the same every game.
* virtual ~Socrates(): included just out of custom (function body does nothing)
* virtual void doSomething(): If dead, Socrates doesn’t do anything. Else, if a key was pressed, he can move counterclockwise (KEY\_PRESS\_LEFT) or clockwise (KEY\_PRESS\_RIGHT) by 5 degrees. Also, he can shoot a spray directly in front of him if he has any left (KEY\_PRESS\_SPACE) or 16 flames in 22 degree increments from the direction he’s facing (KEY\_PRESS\_ENTER). If he didn’t press a key, his spray count starts to increase
* virtual void damage(int hp): If he’s hurt but still alive, the SOUND\_PLAYER\_HURT is played. Else, he’s designated as dead, SOUND\_PLAYER\_DIE is played, and the number of lives he has is decreased by 1.
* int getHP(): returns Socrates’ health points for StudentWorld to display
* int getSpray(): returns Socrates’ spray count for StudentWorld to display
* int getFlame(): returns Socrates’ flamethrower charge count for StudentWorld to display
* void restoreHealth(): sets Socrates’ health points to 100 (as in the case he picks up a RestoreHealthGoodie)
* void increaseFlame(): increases Socrates’ flame count by 5 (as in the case he picks up a FlamethrowerGoodie)

Testing getHP(), getSpray(), getFlame(), restoreHealth(), increaseFlame() was quite straightforward, as I compared the printed values of these member variables along with calling the methods. I tested the damage(int hp) function with values less than 100 (that would only hurt Socrates) and those greater than or equal to 100 (that would kill him) to make sure the various functionality works in different situations. For doSomething(), the move functionality was tested by comparing expected coordinates and orientation for Socrates with the output. The initial positions and orientations of the generated Sprays and Flames were also tested.

1. Dirt: public Actor

* Dirt(StudentWorld\* sw, double startX, double startY): initializes Dirt object at the location
* virtual ~Dirt(): included just out of custom (function body does nothing)
* virtual void doSomething(): Dirt really doesn’t do anything, so the function body is empty

Testing the Dirt class doesn’t really make sense without its interactions with other objects. As the Bacteria and Projectile classes were written, the Dirt functionality was fully tested. Dirt piles should block the movement of Bacteria, which was indeed observed to be the case in test scenarios where the Dirt pile lay between a Bacteria object and Socrates or Food. Similarly, Dirt piles should be destroyed when hit by a Projectile, which was also observed to be the case when firing Sprays and Flames at individual Dirt piles.

1. Pit: public Actor

* Pit(StudentWorld\* sw, double startX, double startY): initializes Pit at location.
* virtual ~Pit(): included just out of custom (function body does nothing)
* virtual void doSomething(): if the Pit has emitted all its bacteria, it lets the StudentWorld know and sets itself to dead without doing anything else. Otherwise, it has a 1/50 chance of emitting Bacteria each tick. If it is going to emit a Bacteria object during a tick, each type with remaining members has an equal chance of being emitted. It tells the StudentWorld a bacteria was emitted and plays SOUND\_BACTERIUM\_BORN.

I tested that the Pit class emitted the right types of bacteria by tracking the respective counts of each type with the StudentWorld setGameStatText function. This helped me make sure the right amounts of each type were eventually released (5 Salmonella, 3 AggressiveSalmonella, and 2 Ecoli). I also displayed the counts of Pits and Bacteria the StudentWorld had, as it would be necessary to advance to the next level when both were 0.

1. Bacteria: public Actor

* Bacteria(StudentWorld\* sw, double startX, double startY, int imageID, int damage, int hp, int hurtSound, int deadSound): Since the Bacteria class could initialize Salmonella, AggressiveSalmonella, or Ecoli, values for imageID, damage, hp, hurtSound, and deadSound vary by type and are included as input parameters.
* virtual ~Bacteria: included just out of custom (function body does nothing) and also to ensure derived classes properly destruct this
* virtual void doSomething(): If dead, nothing is done. Next, a specialized action (private pure virtual function) for the Bacteria is called. After that, if Socrates and the Bacteria are touching, Socrates is damaged by the appropriate amount of damage for the Bacteria. Else, if the Bacteria ate 3 Food objects, it generates a clone (also pure virtual function, as the new object to generate will vary). Otherwise, if the Bacteria is touching a Food object it eats it. Finally, a second specialized action (private pure virtual function again) for the Bacteria is called.
* virtual void damage(int hp): Actor’s default function was overwritten here, which takes into account the fact Bacteria have hp. If the Bacteria is hurt but still alive, the hurtSound is played. Otherwise, the Bacteria calls setDead(), the deadSound is played, the score is increased by 100, 50% of the time a Food object is created, and StudentWorld is told the Bacteria count should decrease by 1.
* int getPlan() const: returns the Bacteria’s planned distance so derived classes can use its value
* void setPlan(int plan): sets the Bacteria’s planned distance to the input parameter

Because of the presence of 3 pure virtual functions in Bacteria (the two specialized actions and cloning), the class was abstract and could not be tested until derived classes were defined. getPlan() and setPlan() were both trivial. The core functionality of doSomething() was verified with each unique Bacteria type, ensuring that at each tick, they would check for Socrates, having eaten enough food to clone, or more food to eat. This was done by initializing 3 food objects and a Bacteria alone, observing that the Bacteria cloned itself in the right position through a printout, and witnessing it damage Socrates by the correct increments over time. Additionally, the damage functionality was tested by firing Sprays and Flames at each Bacteria type.

1. Salmonella: public Bacteria

* Salmonella(StudentWorld\* sw, double startX, double startY): initializes Salmonella at the location with damage value of 1 and 4 hp
* virtual ~Salmonella(): included just out of custom (function body does nothing)

doSomething() captured most of the movement logic for the Salmonella. In fact, its first specialized action was to do nothing. However, its next specialized action was to try moving in the current direction, or if not then toward the closest food, or if neither applied, move randomly. This was tested by placing Food nearby the Salmonella when it was initialized and having the Salmonella with an initial movement plan more than 0 (the Salmonella continued in its current unblocked direction rather than pursue the Food).

1. AggressiveSalmonella: public Bacteria

* AggressiveSalmonella(StudentWorld\* sw, double startX, double startY): initializes AggressiveSalmonella at the location with damage value of 2 and 5 hp
* virtual ~AggressiveSalmonella(): included just out of custom (function body does nothing)

Again, doSomething() was mostly defined by the Bacteria base class, but the specialized actions were also written. The first checked if Socrates was within 72 pixels and attempted to move toward him (possibly getting stuck). The second would do nothing if Socrates was that close; otherwise, it would lead the AggressiveSalmonella to attempt continuing to move in its current direction, or if not then toward the closest food, or if neither applied, move randomly. These additional movement definitions for AggressiveSalmonella were tested by having a Dirt object directly between Socrates and the Bacteria and observing the Bacteria get stuck. In addition, a Food object was added nearby to ensure the Bacteria still did this. Finally, a Food object was placed close to the AggressiveSalmonella with Socrates far away to watch the Bacteria move toward the Food.

1. Ecoli: public Bacteria

* Ecoli(StudentWorld\* sw, double startX, double startY): initialized Ecoli at the location with damage value of 4 and 5 hp
* virtual ~Ecoli(): included just out of custom (function body does nothing)

Just like the other two Bacteria, the base code for doSomething() controls most of the movement logic for Ecoli. Like Salmonella, Ecoli doesn’t have a first specialized action. However, its second specialized action is like AggressiveSalmonella, only even more aggressive. No matter where Socrates is, Ecoli moves toward him. If the Ecoli gets stuck, it tries to change its direction to get around an obstacle. However, Ecoli can still get stuck. This was tested by first placing one Dirt object between Socrates and the Ecoli. The Ecoli was able to successfully move around it and attack Socrates. Then, three Dirt objects were placed between the two, in a way that the Ecoli would be “baited” and get stuck trying to attack Socrates.

1. Chance: public Actor

* Chance(StudentWorld\* sw, double startX, double startY, int imageID, int scoreIncrease, bool goodie): initializes a Fungus or Goodie at the location with the right imageID, scoreIncrease (-50 for Fungus and +100 for Goodies), and whether the Chance object is a Goodie
* virtual ~Chance(): included just out of custom (function body does nothing) and also to ensure derived classes properly destruct this
* virtual void doSomething(): if not alive, do nothing. Else, if overlapping with Socrates, adjust the score by the scoreIncrease and set self to dead. If this is a Goodie, play SOUND\_GOT\_GOODIE. Finally, do the specialized Chance action (private pure virtual function). If not overlapping with Socrates, check if the lifetime has expired (set dead if so). If still alive, decrease the lifetime by 1 tick.

Chance is also an abstract class and so could not really be tested until derived classes were written. Once they were though, it was confirmed that the score increased by 100 when Socrates picked up a Goodie and decreased by 50 when he encountered a Fungus. Also, the initial lifetime and current lifetime each tick were printed out to ensure proper expiration took place.

1. ExtraLifeGoode: public Chance

* ExtraLifeGoodie(StudentWorld\* sw, double startX, double startY): initializes ExtraLifeGoodie at the location
* virtual ~ExtraLifeGoodie(): included just out of custom (function body does nothing)

The specialized action for ExtraLifeGoodie was to increase the number of lives by 1. This was tested by Socrates walking over an ExtraLifeGoodie and the Game Stat display at the top of the screen reflecting that change.

1. FlamethrowerGoodie: public Chance

* FlamethrowerGoodie(StudentWorld\* sw, double startX, double startY): initializes FlamethrowerGoodie at location
* virtual ~FlamethrowerGoodie(): included just out of custom (function body does nothing)

The specialized action here is increasing Socrates’ flame by 5. This was similarly tested by Socrates walking over a FlamethrowerGoodie and the Game Stat display at the top of the screen reflecting that change in the flame count.

1. RestoreHealthGoodie: public Chance

* RestoreHealthGoodie(StudentWorld\* sw, double startX, double startY): initializes RestoreHealthGoodie at location
* virtual ~RestoreHealthGoodie(): included just out of custom (function body does nothing)

This specialized action restores Socrates’ health to 100. This was similarly tested by Socrates walking over a RestoreHealthGoodie and the Game Stat display at the top of the screen reflecting that change in the hp.

1. Fungus: public Chance

* Fungus(StudentWorld\* sw, double startX, double startY): initializes RestoreHealthGoodie at location
* virtual ~Fungus(): included just out of custom (function body does nothing)

A Fungus’ specialized action is damaging Socrates by 20. This was similarly tested by Socrates walking over a Fungus and the Game Stat display at the top of the screen reflecting that change in the hp.

1. Projectile: public Actor

* Projectile(StudentWorld\* sw, int imageID, double startX, double startY, Direction dir, int maxTravel, int damage): initializes a Projectile with the appropriate image, location, direction, maximum travel distance, and damage capability
* virtual ~Projectile(): included just out of custom (function body does nothing)
* virtual void doSomething(): If dead, don’t do anything. Else, if the Projectile overlaps with an Actor that is both alive and targetable, the Actor is damaged by the appropriate amount and the Projectile is set to dead. If nothing appropriate overlaps, the Projectile moves in its current direction by SPRITE\_WIDTH pixels. If it’s reached its maximum distance, it’s set to dead.

Projectile is not an abstract class, but no object should ever be created for our game (except for in derived Sprays and Flames). (I made the design assumption that no one would tamper with this, and it would not need to be hardcoded.) The functionality was tested with those classes, by printing the position of the Projectile each tick and ensuring it could destroy Dirt, Chance objects, and Bacteria. It was confirmed that the Projectile died after travelling the maximum distance.

1. Flame: public Projectile

* Flame(StudentWorld\* sw, double startX, double startY, Direction dir): initializes a Flame at the location, with the direction, maximum travel distance of 32, and damage of 5
* virtual ~Flame(): included just out of custom (function body does nothing)

Flame was tested by ensuring the maximum travel distance was valid and objects died after the appropriate number of ticks. In addition, Flame’s high damage level meant every Bacteria was killed with one shot, not to mention the Dirt and Goodie without hp being destroyed with one hit as well.

1. Spray: public Projectile

* Spray(StudentWorld\* sw, double startX, double startY, Direction dir): initializes a Spray at the location, with the direction, maximum travel distance of 112, and damage of 2
* virtual ~Spray(): included just out of custom (function body does nothing)

Spray was also tested by ensuring the maximum travel distance was valid and objects died after the appropriate number of ticks. Spray’s relatively lower damage level meant each Bacteria took at least two hits to kill (AggressiveSalmonella and Ecoli took three). Dirt and Goodie without hp were still destroyed with one hit.

1. Food: public Actor

* Food(StudentWorld\* sw, double startX, double startY): initializes Food at that location
* virtual ~Spray(): included just out of custom (function body does nothing)
* virtual void doSomething(): does nothing

Food, like Dirt, can’t really be tested without interactions with other classes. The Bacteria functionality, cloning after eating 3 Food objects, was thoroughly tested as described earlier. Also, it was confirmed during testing that Food could not be damaged by Projectiles and would not block movement.