1. **High-level description of each of my public member functions.**

**In class Actor**

Actor(StudentWorld\* stdPtr, int imageID, double startX, double startY,

Direction dir, int depth, double size)

:GraphObject(imageID, startX, startY, dir, depth, size), m\_stdPtr(stdPtr),

m\_passable(false), m\_isAlive(true), m\_canBeBurned(true),

m\_canUseExit(false), m\_canBeInfected(true),m\_canStepOnLandmine(false),

m\_explosive(false), m\_blockFire(false), m\_canFallIntoPit(false),

m\_isInfected(false),m\_canHoldVaccine(false), m\_numVaccines(0),

m\_ticksPassed(0)

{

}

-It is a constructor of Actor class. It initializes Actor objects with according information that are sent as parameters like imageID, startX and startY. Since Actor is derived from its base class GraphObject, the constructor also calls GraphObject’s constructor and sends appropriate arguments. Since it’s a constructor, I didn’t make it virtual.

virtual ~Actor(){}

-It is a destructor of Actor class and I made it virtual to prevent future possible memory leaks when polymorphism is applied.

virtual void doSomething()=0;

-This function lets each derived class of Actor to do something during each tick by overriding this function. Since those derived classes will behave different ways as well as we cannot determine what the Actor itself would do during each tick, I set it to pure virtual function.

bool getPassable(){return m\_passable;}

-It is a getter function that returns the bool value ‘m\_passable’ which represents if another object can pass/intersect this Actor object’s bounding box. This function is defined in Actor class so that it can be used in StudentWorld class when StudentWorld does work using Actor pointers. Since all getter functions would behave similarly (only return appropriate member variable), I set them all non-virtual.

bool getIsAlive(){return m\_isAlive;}

-It is a getter function that returns the bool value representing the status of Actor object, whether it is alive or dead.

void setIsAlive(bool status){m\_isAlive = status;}

-It is a setter function that sets the bool value ‘m\_isAlive’ to the parameter(status) sent to the function. Since each type of Actors sets member variables in similar way, I did not set it virtual.

bool getCanBeBurned(){return m\_canBeBurned;}

-It is a getter function that returns a bool value that represents whether an Actor object can be burned by flame.

bool getCanUseExit(){return m\_canUseExit;}

-It is a getter function that returns a bool value that represents whether an Actor object can use the exit.

bool getCanBeInfected(){return m\_canBeInfected;}

-It is a getter function that returns a bool value that represents whether an Actor object can be infected by zombie’s vomit attack.

bool getCanStepOnLandmine(){return m\_canStepOnLandmine;}

-It is a getter function that returns a bool value that represents whether an Actor object can step on a landmine and trigger the mine to explode.

bool getExplosive(){return m\_explosive;}

-It is a getter function that returns a bool value that represents whether an Actor object is explosive.

bool getBlockFire(){return m\_blockFire;}

-It is a getter function that returns a bool value that represents whether an Actor object can block fire from appearing.

bool getCanFallIntoPit(){return m\_canFallIntoPit;}

-It is a getter function that returns a bool value that represents whether an Actor object can possibly fall into pit.

void setCanFallIntoPit(bool isPossible){m\_canFallIntoPit = isPossible;}

-It is a setter function that sets a bool value that represents whether an Actor object can possibly fall into pit.

void setIsInfected(bool infectionStatus){m\_isInfected = infectionStatus;}

-It is a setter function that sets a bool value that represents whether an Actor object is already infected by vomit.

bool getCanHoldVaccine(){return m\_canHoldVaccine;}

-It is a getter function that returns a bool value that represents whether an Actor object can hold vaccine goodie.

int getNumVaccines(){return m\_numVaccines;}

-It is a getter function that returns a integer value that represents the number of vaccines an Actor object currently holds.

void changeNumVaccines(int num){m\_numVaccines += num;}

-It is a setter function that adds or subtracts the number of vaccines an Actor object is holding.

**In class Wall**

Wall(StudentWorld\* stdWorld, double startX, double startY)

:Actor(stdWorld, IID\_WALL, startX, startY, right, 0, 1.0)

{

setCanBeBurned(false);

setCanBeInfected(false);

setBlockFire(true);

}

-It is a constructor and since Wall is derived class of Actor, it also calls Actor’s constructor via initializer list. It also adds adds/modifies some specific characteristics of a wall such as wall cannot be burned by flame or infected by a vomit.

virtual ~Wall(){}

-It is a destructor of Wall and set it to virtual to prevent possible memory leak due to polymorphism.

void doSomething(){}

-It is a function that is overridden in Wall. Although the wall does nothing during each tick, this function must be implemented to create Wall object since doSomething is a pure virtual function in Wall’s base class, Actor.

**In class Exit**

Exit(StudentWorld\* stdWorld, double startX, double startY)

:ActivatingObject(stdWorld, IID\_EXIT, startX, startY, right, 1, 1.0){

setPassable(true);

setCanBeBurned(false);

setCanBeInfected(false);

setBlockFire(true);

}

-It is a simple constructor and adds some traits that exits have.

virtual ~Exit(){}

-A destructor of Exit class.

void doSomething();

-A function that is called every tick to do specific tasks assigned to Exit.

**In class Pit**

Pit(StudentWorld\* stdWorld, double startX, double startY)

:ActivatingObject(stdWorld, IID\_PIT, startX, startY, right, 0, 1.0)

{

setCanBeBurned(false);

setCanBeInfected(false);

setPassable(true);

}

- It is a simple constructor and adds some traits that Pit objects have.

virtual ~Pit(){}

-A destructor of Pit class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Pit.

**In class Flame**

Flame(StudentWorld\* stdWorld, double startX, double startY, Direction dir)

:ActivatingObject(stdWorld, IID\_FLAME, startX, startY, dir, 1, 1.0)

{

setCanBeInfected(false);

setCanBeBurned(false);

setPassable(true);

setCanStepOnLandmine(true);

}

- It is a simple constructor and adds some traits that Flame objects have.

virtual ~Flame(){}

-A destructor of Flame class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Flame.

**In class Vomit**

Vomit(StudentWorld\* stdWorld, double startX, double startY, Direction dir)

:ActivatingObject(stdWorld, IID\_VOMIT, startX, startY, dir, 0, 1.0)

{

setCanBeInfected(false);

setCanBeBurned(false);

setPassable(true);

}

- It is a simple constructor and adds some traits that Vomit objects have.

virtual ~Vomit(){}

-A destructor of Vomit class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Vomit.

**In class Landmine**

Landmine(StudentWorld\* stdWorld, double startX, double startY)

:ActivatingObject(stdWorld, IID\_LANDMINE, startX, startY, right, 1, 1.0),

m\_isActive(false), m\_safetyTicks(30)

{

setPassable(true);

setCanBeInfected(false);

setCanBeBurned(true);

setExplosive(true);

}

- It is a simple constructor and adds some traits that Landmine objects have.

virtual ~Landmine(){}

-A destructor of Landmine class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Landmine.

**In class Goodie**

Goodie(StudentWorld\* stdWorld,int imageID, double startX, double startY,

Direction dir, int depth, double size)

:ActivatingObject(stdWorld, imageID, startX, startY, dir, depth, size){

setCanBeBurned(true);

setCanBeInfected(false);

setPassable(true);

}

- It is a simple constructor and adds some traits that Goodie objects have.

virtual ~Goodie(){}

-A destructor of Goodie class

**In class VaccineGoodie**

VaccineGoodie(StudentWorld\* stdWorld, double startX, double startY)

:Goodie(stdWorld, IID\_VACCINE\_GOODIE, startX, startY, right, 1, 1.0)

{}

- It is a simple constructor and adds some traits that VaccineGoodie objects have.

virtual ~VaccineGoodie(){}

-A destructor of VaccineGoodie class

void doSomething();

-A function that is called every tick to do specific tasks assigned to VaccineGoodie.

**In class GasCanGoodie**

GasCanGoodie(StudentWorld\* stdWorld, double startX, double startY)

:Goodie(stdWorld, IID\_GAS\_CAN\_GOODIE, startX, startY, right, 1, 1.0)

{

}

- It is a simple constructor and adds some traits that GasCanGoodie objects have.

virtual ~GasCanGoodie(){}

-A destructor of GasCanGoodie class

void doSomething();

-A function that is called every tick to do specific tasks assigned to GasCanGoodie.

**In class LandmineGoodie**

LandmineGoodie(StudentWorld\* stdWorld, double startX, double startY)

:Goodie(stdWorld, IID\_LANDMINE\_GOODIE, startX, startY, right, 1, 1.0)

{

}

- It is a simple constructor and adds some traits that LandmineGoodie objects have.

virtual ~LandmineGoodie(){}

-A destructor of LandmineGoodie class

void doSomething();

-A function that is called every tick to do specific tasks assigned to LandmineGoodie.

**In class Agent**

Agent(StudentWorld\* stdWorld,int imageID, double startX, double startY,

Direction dir, int depth, double size)

:Actor(stdWorld, imageID, startX, startY, dir, depth, size)

{

setCanStepOnLandmine(true);

setCanFallIntoPit(true);

}

- It is a simple constructor and adds some traits that Agent objects have.

virtual ~Agent(){}

-A destructor of Agent class

**In class Human**

Human(StudentWorld\* stdWorld, int imageID, double startX, double startY)

:Agent(stdWorld, imageID, startX, startY, right, 0, 1.0),

m\_infectionCount(0), m\_playedSoundInfected(false)

{

setCanUseExit(true);

setCanHoldVaccine(true);

}

- It is a simple constructor and adds some traits that Human objects have.

virtual ~Human(){}

-A destructor of Human class

int getInfectionCount(){return m\_infectionCount;}

- It returns the infection count of a human object. If a human is infected, its infection count increases by one every tick. This function is used to display Penelope’s infection status every tick and when determining whether an infected human becomes a zombie or not.

**In class Penelope**

Penelope(StudentWorld\* stdWorld, double startX, double startY)

:Human(stdWorld, IID\_PLAYER, startX, startY), m\_numLandmines(0),

m\_numFlames(0)

{

setIsAlive(true);

}

- It is a simple constructor of Penelope.

virtual ~Penelope(){}

-A destructor of Penelope class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Penelope. If the Penelope is not dead, it can either move to a certain direction or use goodies if she has some. All these actions are determined by the user’s key stroke.

int getNumLandmines(){return m\_numLandmines;}

-A getter function that returns the number of landmines Penelope currently holds.

int getNumFlames(){return m\_numFlames;}

-A getter function that returns the number of flames Penelope currently holds.

void changeNumLandmines(int num){m\_numLandmines += num;}

-A setter function that updates the number of landmines Penelope has by the parameter value num.

void changeNumFlames(int num){m\_numFlames += num;}

-A setter function that updates the number of flames Penelope has by the parameter value num.

**In class Citizen**

Citizen(StudentWorld\* stdWorld, double startX, double startY)

:Human(stdWorld, IID\_CITIZEN, startX, startY), dist\_p(1000), dist\_z(1000)

{

}

- It is a simple constructor of Citizen class.

virtual ~Citizen(){}

-A destructor of Citizen class

void doSomething();

-A function that is called every tick to do specific tasks assigned to Citizen.

**In class Zombie**

Zombie(StudentWorld\* stdWorld, int imageID, double startX, double startY)

:Agent(stdWorld, imageID, startX, startY, right, 0, 1.0), m\_vomitX(-1),

m\_vomitY(-1), m\_movementPlan(0) /\*, m\_destX(-1), m\_destY(-1)\*/

{

setCanBeInfected(false);

}

- It is a simple constructor and adds a trait that Zombie objects have.

virtual ~Zombie(){}

-A destructor of Zombie class

double getVomitX(){return m\_vomitX;}

-A getter function that returns the possible vomit’s x position.

double getVomitY(){return m\_vomitY;}

-A getter function that returns the possible vomit’s y position.

**In class DumbZombie**

DumbZombie(StudentWorld\* stdWorld, double startX, double startY)

:Zombie(stdWorld, IID\_ZOMBIE, startX, startY)

{

setCanHoldVaccine(true);

int holdsVaccine = 1;

int holdOrNot = randInt(1, 10);

if(holdOrNot == holdsVaccine){

changeNumVaccines(1);

}

}

- It is a simple constructor and add traist that DumbZombie objects have.

virtual ~DumbZombie(){}

-A destructor of DumbZombie class

void doSomething();

-A function that is called every tick to do specific tasks assigned to DumbZombie.

**In class SmartZombie**

SmartZombie(StudentWorld\* stdWorld, double startX, double startY)

:Zombie(stdWorld, IID\_ZOMBIE, startX, startY)

{

}

- It is a simple constructor of SmartZombie class.

virtual ~SmartZombie(){}

-A destructor of SmartZombie class

void doSomething();

-A function that is called every tick to do specific tasks assigned to SmartZombie.

**In class StudentWorld**

StudentWorld(std::string assetPath);

-A constructor of StudentWorld. Since StudentWorld is a derived class of GameWorld, it also calls GameWorld’s constructor before its body is executed. I did not make it virtual because it is a constructor.

virtual ~StudentWorld(){

cleanUp();

}

-It is a destructor of StudentWorld class and it calls StudentWorld’s cleanUp() function in its body. Same as before in Actor class, since it is a destructor, I made it virtual.

virtual int init();

-It is a function that is called when a new game starts or when a player completes a level or needs to restart a level. Within the function, it loads in a text file that contains map information and locates Actor objects accordingly. If there is an error in the process, it returns appropriate integer constant that represents the error. In order to create a StudentWorld object, this function must be implemented in StudentWorld class because along with move() and cleanUp() function, they were pure virtual functions in base class ‘GameWorld’.

virtual int move();

-It is a function that is called every tick of the game. It first updates the game status which is displayed at the top of the game. Then, it iterates through all the Actor objects and let them do something if they are alive. If the level was completed, it returns appropriate integer constant.

virtual void cleanUp();

-It is a function that is called when Penelope lost a life or has completed the current level. It deletes and frees all the memory that stored Actor objects

bool doesIntersect(Actor\* actorPtr, double destX, double destY);

-It is a getter function that returns a bool value that represents whether any two Actor objects intersect other’s bounding boxes at point (destX, destY). It is used when determining whether an Agent can move to a certain point in the game.

bool doesOverlap(double actorX, double actorY, double otherX, double otherY);

-It is a getter function that returns a bool value that represents whether an overlap occurs between two Actor objects at (actorX, actorY) and (otherX, otherY). It is used when determining situations where Penelope gains goodie or falls into a pit, etc.

void escapeHumans(double exitX, double exitY);

-It is a function that is called from Exit’s doSomething() function. It checks if any Humans overlap with the exit at (exitX, exitY). If there is overlap, then it saves the appropriate human.

bool doesOverlapWithPlayer(double actorX, double actorY);

-It is a function that returns a bool value that represents whether an Actor object at (actorX, actorY) overlaps with the player Penelope. This function then calls the doesOverlap function in the same class to get the result.

void blastFlame();

-It is a function called from Penelope’s doSomething function. When Penelope has at least one number of flames and a user presses space bar, this function blasts three flames at most to the direction the Penelope is facing. However, if there is an Actor object that can block flames like wall, flame does not get created at that spot.

void explodeMine(Actor\* minePtr, bool triggeredByFlame);

-It is a function that explodes a landmine. If the landmine is triggered by flame, or an Actor object overlaps with the mine, the mine explodes. When it explodes, it plays a sound of explosion and introduces 9 flame objects on the spot of landmine and its surroundings. After explosion, the landmine is set to dead so that it will be removed in the following tick. Also a pit is created on the spot where there was a landmine.

void fallIntoPit(Actor\* pitPtr);

-It is a function that determines whether an Actor object overlaps and falls into a pit. It kills any object that falls into a pit. If the killed object was a citizen, we lose points and if it was a zombie, we gain points.

double determineDistToPenelope(Actor\* actorPtr);

-It returns the distance to Penelope object from an Actor object that actorPtr points to. It is used when determining whether a citizen or a smart zombie would follow Penelope or not.

double determineDistToClosestZombie(double citizenX, double citizenY);

-It returns the distance to a closest zombie from a citizen at (citizenZ, citizenY). It is used when a citizen decides to run away if a zombie gets near the citizen.

bool throwVomit(Zombie\* zombiePtr);

-It decides whether a zombie would throw a vomit to a human or not. Based on zombiePtr the function checks if any human overlaps with the zombie’s vomit coordinates. If they overlap, a zombie vomits at human with a 1/3 chance.

bool findClosestPersonAndFollow(Zombie\* zombiePtr);

-This function is used by smart zombie’s doSomething function. It looks for a closest Human from a zombie pointed by zombiePtr. If ta zombie can find a human that is closer or equal to 80 pixels, the zombie will follow the human object and this function will return true. If not, it will just return false.

void followPenelope(Actor\* actorPtr, int moveDistance);

-This function is called by citizen when a citizen tries to follow Penelope.

void citizenBecomeZombie(Actor\* citizenPtr);

-This is called when an infected citizen become a zombie when its infection rate reaches 500. There is 30% chance of becoming a smart zombie and 70% chance of becoming a dumb zombie.

void runaway(Actor\* citizenPtr);

-This is called when a zombie approaches a citizen pointed by citizenPtr. When a zombie comes closer than 80 pixels from the citizen, the citizen will try to run away from the zombie. It will calculate the distance from the approaching zombie after the possible run-away movement. The citizen will choose the option where it leads the citizen to be at the furthest from the approaching zombie. Then the citizen will move 2 pixels based on the option chosen above.

void incrementVaccine();

-It is called when a player earns a vaccine goodie and increments the number of vaccines that the player is holding by one.

void incrementFlameCount();

-It is called when a player earns a gas can goodie and increments the number of flames that the player is holding by five.

void incrementLandmineCount();

-It is called when a player earns a land mine goodie and increments the number of landmines that the player is holding by two.

void plantLandmine();

-It is called from Penelope’s doSomething() function when a player types a tab key and a player holds at least one landmine. It introduces a new landmine object at the position where player is standing.

void damageObjects(Actor\* flamePtr);

-This function is called by Flame’s doSomething() function. If a flame overlaps with an object that can be burned, it will burn and kill the object. If the object is explosive, it will trigger the explodeMine() function. When it kills a dumb zombie that happens to be holding a vaccine goodie, the dumb zombie, as it dies, will fling it away to a position where it does not overlap with any other objects.

void infectObjects(Actor\* vomitPtr);

-This function is called by Vomit’s doSomething() function. The vomit object pointed by vomitPtr checks if it overlaps with any other human objects. If it overlaps, then it sets that human object’s infection status to true.

**2. Functionality that I have failed to finish/ bugs.**

I believe I have debugged most of my issues that I had before by now. I initially had a bug when I planted multiple landmines together and try to explode all at once through shooting a flame at it. The game sometimes suddenly ended throwing an error or continued but the speed became considerably slower and the sound did not came out. However, I debugged it by setting flame object that when it tries to destroy some object, the flame should first check if the overlapping object is already dead. If it is already dead, the damageObject does not damage the dead object again.

**3.A list of other design decisions and assumptions I made.**

- I made a assumption that when a dumb zombie that holds a vaccine dies due to a flame, the vaccine that was thrown should also be damaged by the flame if the vaccine overlaps with the flame. Also, I assumed that when objects have same depth, then we cannot guarantee that one will always appear on top of the other. For example, when I plant a landmine above exit, both landmine and exit objects have a depth of 1, so sometimes landmine shows on top of exit and vice versa. And the demo given also proved to behave like that. Also when more than one landmines are overlapped with each other, if I explode one mine with flame, the flame that gets created by explosion of a landmine triggers other landmines to explode. Also, when a landmine explodes beside a wall, I could observe a flame being created behind the wall for a short amount of time. I assumed it is okay as the demo also showed the same behavior. Plus, if I shoot a flame to a space between the wall bricks, the flame often go through in between the walls. I also assumed it is okay as demo behaved the same.

**4. How I tested each of my classes.**

Class Actor

Since Actor class is the base class of all the objects that are used in the game. It holds bool values that some of the derived classes share the same value in common. This is later used to determine if a certain action can be made. Also, I set the doSomething function of this Actor class to pure virtual since I won’t be creating any ‘Actor’ objects, but all the derived classes of Actor will behave differently by overriding doSomething function accordingly.

Class Wall

I tested three characteristics of wall object. First, I proved that it won’t be damaged by flame by shooting flames to a wall and wall did not disappear. Not only that but also the wall should block the flame being created at that position. Then, I tested wall will not be infected by vomit by alluring smart zombie through Penelope and observed if the zombie would vomit on a wall. I confirmed that the zombie did not vomit onto a wall.

Class Activating Object

This class is a sub class of Actor so I just checked if it is calling the Actor’s constructor correctly before it executes the body of its constructor.

Class Exit

I tested Exit class if Agent objects can pass through and overlap with the exit. Then I shot the flame at it and saw exit was not damaged but blocks flames. Similar to testing a wall, I brought smart zombie near the exit to see if zombie vomits at an exit. It did not. Then I checked if the exit works properly that it only lets Penelope to escape when there are no citizens left to be freed.

Class Pit

I tested Pit class that it cannot be burned by flame or infected by vomit. I let Agents and to walk to it and when Agents overlapped with the pit, they died as expected. Also, I planted a number of landmines and shot a flame to it to test the same number of pits are being created as landmines explodes.

Class Flame

I tested Flame class by shooting flames to other objects. As planned, it only damaged and killed the objects that are vulnerable to flame attacks. Flame triggered the landmines to explode when they overlapped and the flame was blocked by a wall or an exit object when I shot them.

Class Vomit

I tested Vomit class by taking Human objects near zombies and see what happens when a zombie vomits. I could confirm that when a zombie vomits by 1/3 chance, the infection status of Human object that got attacked became true and increased by one every tick.

Class Landmine

I tested landmine by stepping on it with both Humans and Zombies. In both cases the landmine exploded resulting nine flames occur and a pit after. I also tested the score gains and losses depending on who died from the explosion and it worked fine.

Class Goodie

I tested Goodie objects if they can be burned by flame and can be overlapped with others. It showed that when a flame overlaps with a Goodie, the goodie gets destroyed. Also when Penelope overlaps with the goodie, the player earned the according goodie and the count went up as well.

Class VaccineGoodie

I tested this Class by using it after being infected by zombie’s vomit. As soon as I used the vaccine goodie, the infection rate stopped increasing and became 0 as expected. Also, I intentionally set the percentage of dumb zombie to must hold a goodie and then killed the zombie to see if a vaccine goodie is dropped appropriately.

Class GasCanGoodie

I tested GasCanGoodie by moving Penelope to overlap with it and see if the user can earn flames from it. As expected, Penelope earned five flame shots as soon as it overlapped with the gas can goodie.

Class LandmineGoodie

I tested LandmineGoodie by moving Penelope to overlap with it and see if the user can earn landmine from it. As expected, Penelope earned 2 landmines as soon as it overlapped with the landmine goodie.

Class Agent

Since it is the base class of both Humans and Zombies, I added some feature that both of them share in common such as they can step on a landmine and fall into a pit. Then I checked if the derived class has appropriate features when constructed.

Class Human

Since it is a base class of Penelope and Citizen, I added some features that both of them share in common. Since both Penelope and Citizen can use exit I set it true under the constructor and checked it they can actually use the exit when constructed.

Class Penelope

I tested Penelope’s class by checking the infection status at the beginning of each tick. If it was infected, I then checked if the infection count rose by one in each tick. Then I also checked if Penelope behaved appropriately according to the key stroke the user made. Plus, I confirmed that it does not shoot a flame or plant a mine if she does not have any.

Class Citizen

I tested Citizen class by checking if the citizen moves away from a zombie approaching. I changed the level1.txt map so that smart zombie will be near the citizen and set Penelope far from the citizen. As expected, the citizen ran away from the smart zombie and smart zombie tried to catch the citizen. Also, when a citizen dies due to infection, I checked to see if it properly becomes a zombie after it dies.

Class Zombie

Since it is a base class of DumbZombie and SmartZombie, I added some features that both of them share in common. Thus I set they cannot be infected by vomit and confirmed the result by placing two zombies next to each other. As planned, they did not vomit to each other.

Class DumbZombie

I tested dumb zombies if they made new movement plans when it becomes zero. Also, checked if they hold a vaccine goodie by 10% chance by killing many dumb zombies. I tested if dumb zombies would fall into a pit and can be damaged by flame by controlling Penelope.

Class SmartZombie

I tested smart zombies similar to dumb zombies. In addition, I checked if the smart zombie follows a Human properly if it gets near the Human objects. I brought both Penelope and Citizen one at a time and the smart zombie followed the Human object as expected.

Class StudentWorld

I first tested StudentWorld class successfully implements the three functions: init, move and cleanup. Since those were the core functions of StudentWorld, I checked if the init is called appropriately when a new game starts. I compared the level text file with the actual game screen to see if there are any differences in between.

Then I tested move. I checked the game status was showing the correct values as game went on by earning and losing goodies and points. I confirmed it returns correct return values so that every time when the game ends, the screen is displaying correct messages.

The cleanup function frees all the pointers and memory that were allocated in the vector actorList. After it frees them, I checked the vector’s size is equal to zero so test cleanup function’s behavior.