Project3 Report

1. A high-level description of public member functions in each classes

**class Actor : public GraphObject**

**{**

**public:**

**Actor(int imageID, double startX, double startY, StudentWorld\* sp, int startDirection = right, int depth = 0);**

**Description:** constrctor of Actor that initializes its base class GraphObject and its StudentWorld pointer.

**StudentWorld\* getWorld() const;**

**Description:** Get this actor's world, every Actor lives in a World and its implementation for all derived classes should be the same, so it’s non-virtual.

**virtual void doSomething() = 0;**

**Description:** Every Actor should have a chance to perform for each tick, and its implementation is different for every Actor class and should be determined at run time. Also there is no implementation for the base Actor class, so pure virtual.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** Activate the object and perform its effect on another Actor a, its implementation is different for every Actor class and should be determined at run time. so virtual.

**virtual bool isDead() const;**

**Description:** determine if whether this actor is dead. Every Actor should either be dead or alive, and Haslife classes implement this differently (instead of returning false immediately), so virtual.

**virtual void setDead();**

**Description:** For general Actors it does nothing, and for Haslife it sets the character dead. Different implementations for different classes so virtual.

**virtual void changeInfectionStatus(bool b);**

**Description:** If this object can be infected by vomit, change its InfectionStatus. It has different implementation for Agent and other classes, so virtual.

**virtual bool blockMovement() const;**

**Description:** Does this object block agent movement? It returns true or false depending on the classes so it is virtual

**virtual bool blockFlame() const;**

**Description:** Does this object block flames? It returns true or false depending on the classes so it is virtual

**virtual bool isInfected() const;**

**Description:** Is this object infected by Vomit? It implements differently for Agent classes and other classes so it is virtual.

**virtual bool needSave() const;**

**Description:** Does this object need to be saved by Penelope? It returns true for citizen and false for other classes so it is virtual.

**virtual bool canExplode() const;**

**Description:** Can this object explode? It returns true for landmine and false for other classes so it is virtual.

**virtual bool canVomit() const;**

**Description:** Can this object vomit? It returns true for Zombie(DumbZombie and SmartZombie) and false for other classes, so it is virtual.

**virtual bool isHot() const;**

**Description:** Is this object hot? It returns true for flame and false for other classes, so it is virtual.

**virtual bool canDropVaccine() const;**

**Description:** Can this object drop vaccine? It returns true for Dumbzombie and false for other classes, so it is virtual.

**virtual bool attractCitizens() const;**

**Description:** Does this object attract citizens? It returns true for Penelope and false for other classes, so it is virtual.

**virtual bool collectible() const;**

**Description:** Is this object collectible by penelope? It returns true for all kinds of goodies and false for other classes, so it is virtual.

**}**

**class Haslife : public Actor**

**{**

**public:**

**Haslife(int imageID, double startX, double startY, StudentWorld\* sp, int startDirection = right, int depth = 0);**

**Description:** constructor of Haslife that initializes its base class Actor and set m\_isDead to false;

**virtual void setDead();**

**Description:** For every object of a class derived from Haslife, set this object to dead, so it is defined in the Haslife class. It inherits from Actor class so it is virtual.

**virtual bool isDead() const;**

**Description:** return whether this object is dead or not, for every object of a class derived from Haslife, it returns the value of m\_isDead, so it is defined in the Haslife class. It inherits from the virtual function in Actor class so it is virtual.

**}**

**class Movable : public Haslife**

**{**

**public:**

**Movable(int imageID, double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Movable that initializes it base class Haslife.

**virtual bool blockMovement() const;**

**Description:** It returns true for every Movable object because they block movement of others. It inherits from the virtual function in Actor class so it is virtual.

**void setDestination(double& dest\_x, double& dest\_y, Direction dir, int len);**

**Description:** It sets dest\_x and dest\_y according to dir and len. Its derived classes should set its destination in the same way so it is defined as non-virtual.

**};**

**class Agent : public Movable**

**{**

**public:**

**Agent(int imageID, double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Agent that initializes its base class Movable, sets its infectionStatus to false and infectionCount to 0.

**virtual void changeInfectionStatus(bool b);**

**Description:** Set this object’s infectionStatus to b. It is inherited from the virtual function in Actor so it is virtual.

**virtual bool isInfected() const;**

**Description:** return this object’s infectionStatus. It is inherited from the virtual function in Actor so it is virtual.

**int infectionCount();**

**Description:** return this object’s infectionCount. It behaves the same way for all its derived classes so it is non-virtual.

**void setInfectionCount(int num);**

**Description:** Set this object’s infectionCount to num. It behaves the same way for all its derived classes so it is non-virtual.

**void increaseInfectionCount(int amount);**

**Description:** increase this object’s infectionCount by amount. It behaves the same way for all its derived classes so it is non-virtual.

**};**

**class Wall : public Actor**

**{**

**public:**

**Wall(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Wall that initializes it base class Actor.

**virtual void doSomething();**

**Description:** It does nothing for this object. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool blockMovement() const;**

**Description:** It returns true for every Wall object because they block movement of others. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool blockFlame() const;**

**Description:** It returns true for every Wall object because they block flames. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Penelope : public Agent**

**{**

**public:**

**Penelope(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Penelope that initializes it base class Agent.

**virtual void doSomething();**

**Description:** get input and move and using Goods accordingly. It inherits from the virtual function in Actor class so it is virtual.

**void pickUpGoodieIfAppropriate(Goodie\* g);**

**Description:** Let g give penelope some utilities according to its type. This function is only used by penelope to pick up goodies so it is non-virtual.

**virtual bool attractCitizens() const;**

**Description:** returns true because penelope attracts citizens. It inherits from the virtual function in Actor class so it is virtual.

**void increaseVaccines(int amount);**

**Description:** increase the Vaccine count by amount. Only used by Penelope class so non-virtual

**void increaseFlameCharges(int amount);**

**Description:** increase the FlameCharge Count by amoumt. Only used by Penelope class so non-virtual

**void increaseLandmines(int amount);**

**Description:** Increase the number of landmines the object has. Only used by Penelope class so non-virtual

**int getNumVaccines() const;**

**Description:** get number of Vaccines this object has. Only used by Penelope class so non-virtual

**int getNumFlameCharges() const;**

**Description:** get number of FlameCharges this object has. Only used by Penelope class so non-virtual

**int getNumLandmines() const;**

**Description:** get number of Landmines this object has. Only used by Penelope class so non-virtual

**};**

**class Citizen : public Agent**

**{**

**public:**

**Citizen(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Citizen that initializes it base class Agent.

**virtual void doSomething();**

**Description:** Move according to its distance and direction with penelope and zombie. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool needSave() const;**

**Description:** returns true because citizen needs to be saved. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Zombie : public Movable**

**{**

**public:**

**Zombie(double startX, double smartY, StudentWorld\* sp);**

**Description:** constructor of Zombie that initializes its base class Movable.

**virtual void doSomething();**

**Description:** Move and Vomit appropriately. doSomething() is defined here instead of its derived classes because these two kinds of Zombies do the same thing during a tick except a difference in picking directions. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool canVomit() const;**

**Description:** return true because all Zombies can vomit. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Dumbzombie : public Zombie**

**{**

**public:**

**Dumbzombie(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of DumbZombie that initializes its base class Zombie.

**virtual bool canDropVaccine() const;**

**Description:** returns true because DumbZombie can drop vaccines when it dies. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Smartzombie : public Zombie**

**{**

**public:**

**Smartzombie(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of SmartZombie that initializes its base class Zombie.

**};**

**class Exit : public Actor**

**{**

**public:**

**Exit(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Exit that initializes its base class Actor.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** if it overlap with citizens, the Exit exits these citizens. It inherits from the virtual function in Actor class so it is virtual.

**virtual void doSomething();**

**Description:** it first let all citizens overlapping with it uses the exit, and then if there is no more citizens at this level and penelope overlap with it, it informs the StudentWorld to record levelFinished. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool blockFlame() const;**

**Description:** It returns true because Exit block Flames. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Pit : public Actor**

**{**

**public:**

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

**Description:** constructor of Pit that initializes its base class Actor.

**virtual void doSomething();**

**Description:** it calls activateIfAppropriate(), It inherits from the virtual function in Actor class so it is virtual.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** It kills penelope, citizen, zombies that overlaps with it. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Goodie : public Haslife**

**{**

**public:**

**Goodie(int imageID, double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Goodie that initializes its base class Haslife.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** if it overlaps with penelope, it gives penelope utility. It inherits from the virtual function in Actor class so it is virtual.

**virtual void doSomething();**

**Description:** it calls activateIfAppropriate(), It inherits from the virtual function in Actor class so it is virtual.

**virtual void givePenelopeSomething(Penelope\* p) = 0;**

**Description:** It is defined as pure virtual because all Goodies can give penelope some utility, and each of them give penelope different things.

**};**

**class VaccineGoodie : public Goodie**

**{**

**public:**

**VaccineGoodie(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of VaccineGoodie that initializes its base class Goodie.

**virtual void givePenelopeSomething(Penelope\* p);**

**Description:** it increases penelope’s vaccine count by 1. It inherits from the virtual function in Goodie class so it is virtual.

**};**

**class LandmineGoodie : public Goodie**

**{**

**public:**

**LandmineGoodie(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of LandmineGoodie that initializes its base class Goodie.

**virtual void givePenelopeSomething(Penelope\* p);**

**Description:** it increases penelope’s landmine count by 2. It inherits from the virtual function in Goodie class so it is virtual.

**};**

**class GascanGoodie : public Goodie**

**{**

**public:**

**GascanGoodie(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of gascanGoodie that initializes its base class Goodie.

**virtual void givePenelopeSomething(Penelope\* p);**

**Description:** it increases penelope’s flame count by 5. It inherits from the virtual function in Goodie class so it is virtual.

**};**

**class Projectile : public Haslife**

**{**

**public:**

**Projectile(int imageID, double startX, double startY, StudentWorld\* sp, int startDirection);**

**Description:** constructor of Projectile that initializes its base class Haslife, and set its m\_lifespan to 0.

**virtual void doSomething();**

**Description:** it increases its m\_lifespan by 1, if life\_span equals 2, set this object to dead. otherwise, it calls activateOnAppropriateActors(). It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Vomit : public Projectile**

**{**

**public:**

**Vomit(double startX, double startY, StudentWorld\* sp, int startDirection);**

**Description:** constructor of Vomit that initializes its base class Projectile.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** if it overlaps with a citizen or penelope, it changes infectionStatus to true. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Flame : public Projectile**

**{**

**public:**

**Flame(double startX, double startY, StudentWorld\* sp, int startDirection);**

**Description:** constructor of Flame that initializes its base class Projectile.

**virtual bool isHot() const;**

**Description:** returns true because flame is hot (so it will activate landmine immediately). It inherits from the virtual function in Actor class so it is virtual.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** It kills penelope, citizen, zombies that overlaps with it, as well as explode any landmine that overlaps with it. It inherits from the virtual function in Actor class so it is virtual.

**};**

**class Landmine : public Haslife**

**{**

**public:**

**Landmine(double startX, double startY, StudentWorld\* sp);**

**Description:** constructor of Landmine that initializes its base class Haslife.

**virtual void doSomething();**

**Description:** it check if it is active, if not, it decreases its safety tick and return; otherwise, it calls activateOnAppropriateActors(). It inherits from the virtual function in Actor class so it is virtual.

**virtual void activateIfAppropriate(Actor\* a);**

**Description:** it calls explode() if it overlaps with citizen, penelope, zombies or flame. It inherits from the virtual function in Actor class so it is virtual.

**virtual bool canExplode() const;**

**Description:** returns true because landmine can explode. It inherits from the virtual function in Actor class so it is virtual.

**void explode();**

**Description:** sets this object to dead and create eight flames around the landmine and create a pit on the spot of the landmine.

**};**

**class StudentWorld : public GameWorld**

**{**

**public:**

**StudentWorld(std::string assetPath);**

**virtual ~StudentWorld();**

**virtual int init();**

**virtual int move();**

**virtual void cleanUp();**

**// Add an actor to the world.**

**void addActor(Actor\* a);**

**// determine the ability to move**

**bool isBlocked(double dest\_x, double dest\_y, Actor\* itself);**

**// Is creation of a flame blocked at the indicated location?**

**bool isFlameBlockedAt(double x, double y);**

**// get number of**

**int getnCitizens();**

**int getnZombies();**

**// change number of**

**void decrementCitizens();**

**void changenZombies(int n);**

**// Indicate that the player has finished the level if all citizens**

**// are gone.**

**void recordLevelFinishedIfAllCitizensGone();**

**// make penelope pick up goods**

**void penelopePickupGoods(Goodie\* g);**

**// determine the distance with penelope**

**double distanceWithPenelope(Actor\* itself);**

**double distanceWithPenelope(double x, double y);**

**// determine the direction to the nearest agent within 80 pixels**

**bool findNearestAgent(Actor\* itself, Direction& dir1, Direction& dir2);**

**// determine the distance with nearest zombie**

**double findNearestZombie(Actor\* itself);**

**double findNearestZombie(double x, double y);**

**// used by zombie to determine whether it will vomit**

**void findAgentToVomit(Actor\* itself);**

**// get direction to penelope**

**void getDirToPenelope(Actor\* itself, Direction& dir1, Direction& dir2);**

**// create Vaccine Goodie around it somewhere it dies**

**void createVaccineGoodie(double x, double y);**

**// For each actor overlapping a, activate a if appropriate.**

**void activateOnAppropriateActors(Actor\* a);**

**private:**

**// used by various public direction functions**

**void getDirTo(Actor\* itself, Actor\* target, Direction& dir1, Direction& dir2);**

**// compute the distance between two actors**

**double distance(Actor\* p1, Actor\* p2);**

**double distance(double x1, double y1, double x2, double y2);**

**// private data members**

**std::list<Actor\*> li;**

**Penelope\* penelope;**

**int m\_nCitizens;**

**int m\_nZombies;**

**bool levelFinished;**

**};**

**2. A list of all functionality that you failed to finish as well as known bugs in your classes**

**none.**

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

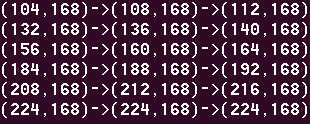
**1.Because the spec does not specify what object would block vomit. When creating vomit my program does not check whether it is legal to create vomit on that spot.**

**2.Because the spec does not say where we can’t put the landmine on, I assume that I could put landmine under the exit.**

**4.** **A description of testing classes**

**1.Penelope:**

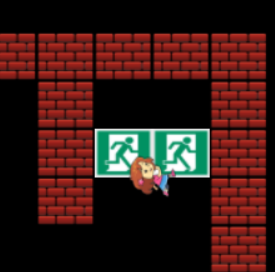
**make sure it does something correct according to the keyboard input: when it is a TAB, SPACE, or ENTER, there is a correct decrement of utilities if the player uses them. and corresponding object or effect is created. When moving, the distance is 4 and it should not move onto walls, citizens, zombies**

**penelope moves 4 pixels in one tick**

**2.Wall:**

**Wall has limited functionality. So I test it by deleting and adding several walls in the level file to make sure it is created in the right place.**

**3.Exit: use cerr to monitor the number of citizens. Move penelope onto the exit, penelope would cover the exit (when the level is not finished). Move citizens to the exit, citizen will be saved and the number of citizens would decrease by 1, score would increase by 500(here I found the exit need to check whether the citizen has died or not, because sometimes citizen would overlap with two exits). When all citizens have been saved(number of citizens is 0), the level finishes when penelope move onto the exit.**

**penelope would cover the exit, because exit has depth 1**

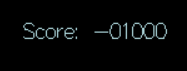
**the number of citizens decreases by 1**

**when there is no citizens on this level and penelope overlaps with exit, level finishes.**

**4.Pit: use cerr to monitor the number of citizens and number of zombies. Move citizens onto the pit, it kills citizen, decreases score by 1000, decreases number of citizen by 1. When dumbzombie move onto it, increase score by 1000, decrease number of zombie by 1, drop vaccine in some cases. When smartzombie move onto it, increase score by 2000, decrease number of zombie by 1. When penelope move onto the pit, it kills her.**

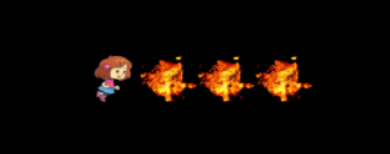
**score of killing a smart zombie**

**killing a zombie decreases number of zombies by 1**

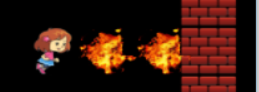
**score of killing a citizen**

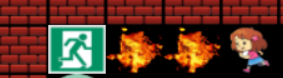
**killing a citizen decreases number of citizens by 1**

**5.Flame: use cerr to monitor the number of citizens and number of zombies. the effect of citizen, zombie and penelope overlapping with it is the same as that in Pit test. When flame overlaps with goodie, it destroys the goodie. When flame overlap with landmine, the landmine explodes immediately. Flame is blocked by exit, wall, so I fire the flame toward exit or wall and found out that it cannot penetrate it.**

**penelope firing the flame**

**flame causes landmine to explode**

**flame blocked by wall**

**flame blocked by an exit**

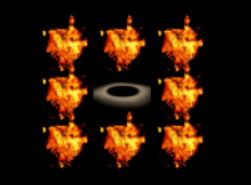
**6.Vomit: use cerr to monitor the infection status and infection count of penelope and citizen. If zombie vomits, because time is not enough for agent to escape, infection status becomes true and infectioncount increases by 1 each tick. When infectionStatus is 500, citizen turns into zombie, which decreases score by 1000, and penelope dies.**

**7.VaccineGoodie/LandmineGoodie/GascanGoodie: use the game stat above to check whether overlapping each goodie would give the correct amount of utilities, as well as the 50 score.**

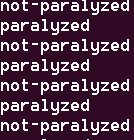


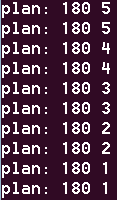
**one vaccine, one landmine, one gascan goodie**

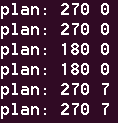
**8.Landmine: overlapping citizen, penelope, zombie onto it would cause it to explode, creating 8 flames around it and a pit at the spot.**

**zombie stepping onto a landmine**

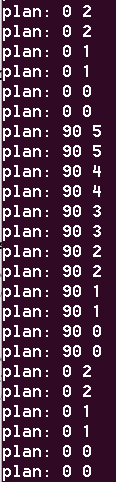
**9.DumbZombie: first check whether it will be paralyzed every 2 ticks. Then use cerr to monitor its moveplan(direction and distance) to make sure it is random)**

**paralyzed every 2 ticks**

**decrease its movement plan by 1 each time**

**changes direction when it moves to wall**

**10.SmartZombie: first check whether it will be paralyzed every 2 ticks. Then use cerr to monitor its moveplan(direction and distance) to make sure it chooses direction closer to agent.**

**when finding a nearby penelope or citizen, it will choose from the two direction taking the zombie closer**

**11.Citizen: First check whether it will be paralyzed every 2 ticks, then use cerr to monitor its distance with penelope and zombie, and its consequent direction of moving to make sure it chooses the correct direction closer to penelope, away from zombie, or none(when no one is nearby)**

