# **High Level Class Member Function Descriptions**

class Actor : public GraphObject

public:

virtual void doSomething() = 0;

virtual bool shouldBlock() const = 0;

virtual bool mortal() const { return false; }

//returns whether an object is considered mortal (can die)

virtual bool canHoldVaccine() const { return false; }

//only smart zombies can hold vaccines

virtual bool infectable() const { return false; }

//returns if an actor can be infected

virtual void infect(bool status) { return; }

//doesn’t do anything for Actor base class

virtual bool infected() const { return true; }

//returns if something is infected (explained later)

virtual void setInfectionStatus() const { return; }

virtual bool flameRetardant() const { return false; }

//returns if object should block flame

bool alive() const { return m\_alive; }

//return alive status

void setAlive(bool status) { m\_alive = status; }

inline StudentWorld\* getWorld() const { return m\_world; }

I made Actor an abstract class (a class with pure virtual functions) because it didn’t make logical sense to have an object that was just an actor. Therefore, I made doSomething and shouldBlock virtual functions, as it didn’t make sense for actors to have common functionality for those functions.

I made a mortal() function in Actor, which returned false, because I wanted a way of differentiating all the zombies and humans from the rest of the actors. The infectable() function is how I differentiated between zombies and citizens. Because I had a separate Penelope pointer, only citizens returned true for infectable(). I also defined flameRetardant() to differentiate the objects that couldn't be damaged by flames (ie. walls, exits and projectiles). I also defined canHoldVaccine() to differentiate between the rest of my actors and smart zombies.

Because my checkVomitVictim() function in StudentWorld iterates through all actors, I had to declare infect() and infected() as such. (Because my checking condition is if(!infected()), I needed all other actors to return true)

I also implemented alive(), setAlive() and getWorld() for all Actors because most actors require a StudentWorld pointer to carry out their functionality, and all Actors need to be able to be checked for alive status.

class StudentWorld : public GameWorld

{

public:

StudentWorld(std::string assetPath);

virtual ~StudentWorld() { cleanUp(); }

virtual int init(); //initializes studentworld object

virtual int move(); //manages game moves

virtual void cleanUp(); //destructor calls this to delete studentworld

void removeDead(); //removes all dead actors

bool canMoveTo(double destX, double destY, Actor\* caller) const;

//checks if movement to destX, destY by Actor caller is blocked

bool canOverlap(double x, double y, Actor\* caller) const;

//checks if something overlaps at x, y with Actor caller

void playerDied();

//handles player dying

void citizenDied(double x, double y);

//handles citizen dying

void citizenSaved();

//handles citizen hitting exit

void zombieDied(double x, double y, Actor\* zombie);

//handles zombie dying

inline void finishLevel() { m\_levelFinished = true; }

//handles level finishing

int getCitizens() const { return m\_numCitizens; }

double closestPerson(double& x, double& y);

//returns the x,y coordinates of the closest person

double closestZombie(double x, double y);

//returns the x,y coordinates of the closest zombie

Actor\* checkOverlap(double x, double y, int& type, Actor\* caller);

//checks if calling object is overlapping with a mortal actor

void incGoodie(int goodie);

//increase penelopes inventory based on goodie picked up

void createZombie(Actor\* caller);

void createFlame(Actor\* caller);

bool checkFlameBlock(double destX, double destY) const;

//check if flame is blocked

void checkFlameDamage(Actor\* caller);

//check if flames damage things

void createMine(Actor\* caller);

void createVomit(double x, double y, Direction dir);

bool checkVomitRange(double x, double y) const;

//check range to see if zombie should vomit

void checkVomitVictims(double x, double y);

//check which actors should be splashed by vomit

void createVaccine(double x, double y);

class Mortal : public Actor

{

public:

virtual bool shouldBlock() const { return true; }

//mortals should block

virtual bool mortal() const { return true; }

//identifier function

virtual bool paralysis() const { return m\_paralysis; }

//zombies and citizens have paralysis and inherit from Mortal so I put it here

void setParalysis(bool status) { m\_paralysis = status; }

I chose to put paralysis() and setParalysis() in the Mortal class because paralysis ticks are shared by both zombies and citizens. Both those classes inherit from Mortal, therefore I decided to set the default function return values in this class.

class Infectable : public Mortal

{

public:

virtual bool infectable() const { return true; }

//penelope and citizens can be infected so I changed this from the Actor class to return true.

virtual void infect(bool status) { m\_infected = status; }

inline int getInfectedTimer() const { return m\_infectedTimer; }

virtual bool infected() const { return m\_infected; }

inline void incInfectionTimer() { m\_infectedTimer++; }

inline void resetInfectionTimer() { m\_infectedTimer = 0; }

I chose to implement all the functions involved in dealing with zombie vomit infection in this class, as both Penelope and citizens inherit from this class. Checking infection status, increasing, and reseting the infection timer are all addressed here.

class Penelope : public Infectable

{

public:

virtual void doSomething();

//implements all of penelope’s actions as outlined in the spec. pseudocode written further below

inline int getVacc() const { return m\_vaccines; }

//inventory checker and setters

inline int getFlames() const { return m\_flameCharges; }

inline int getMines() const { return m\_landMines; }

void incFlames() { m\_flameCharges += 5; }

void decFlames() { m\_flameCharges--; }

void incVacc() { m\_vaccines++; }

void decVacc() { m\_vaccines--; }

void incMine() { m\_landMines += 2; }

void decMine() { m\_landMines--; }

My Penelope class is pretty barebones, as most of the functionality is inherited from subclasses. The only real unique functions independent of the spec’s direct instructions I’ve implemented are Penelope’s inventory setters and getters.

class Wall : public Actor

{

inline virtual void doSomething() { return; }

//walls don’t do anything

virtual bool shouldBlock() const { return true; }

//walls should block movement

virtual bool flameRetardant() const { return true; }

//walls should block flames

class Overlap : public Actor

{

virtual ~Overlap() {}

virtual bool shouldBlock() const { return false; }

virtual void doSomething() = 0;

I declared my Overlap class as pure virtual because nowhere in the game is a pure overlap object created. As a lot of objects in the game are supposed to allow overlapping, yet it doesn’t make sense for an overlapping object in and of itself to do something, I chose to implement doSomething() as pure virtual. Overlapping also inherently necessitates not being able to block movement.

class Exit : public Overlap

virtual bool flameRetardant() const { return true; }

virtual void doSomething();

//checks if level is finished when Penelope is overlapping. (ie. no citizens left) also checks if citizens overlapping to remove from game

class Pit : public Overlap

{

virtual void doSomething();

//checks if anything mortal is overlapping. if so, kill it

class Goodie : public Overlap

{

public:

virtual void doSomething = 0;

virtual bool mortal() const { return true; }

I chose to make the goodie class pure virtual because it doesn’t make sense for an undifferentiated goodie object to do something. Goodies should also return mortal, because their alive state can be changed (when picked up and removed from the game)

class GasCan : public Goodie

virtual void doSomething();

class Vaccine : public Goodie

virtual void doSomething();

class LandMine : public Goodie

virtual void doSomething();

All the goodies do essentially the same thing. Checks if Penelope is overlapping it and if so adds to Penelope’s inventory and removes the goodies

class Mine : public Overlap

{

virtual void doSomething();

//checks if active, if so checks if overlapping anything that could trigger explosion. if so, explode

inline bool active() const { return m\_active; }

inline int getTicks() const { return m\_ticks; }

void decTicks() { m\_ticks--; }

void setActive() { m\_active = true; }

The mine class essentially handles all of its own functionality. Checking/setting ticks, checking if it should explode.

class Projectile : public Overlap

{

virtual void doSomething() = 0;

virtual bool flameRetardant() const { return true; }

void incTick() { m\_tick++; }

inline int getTick() const { return m\_tick; }

I decided to make Projectile pure virtual because it acts as a base class that will never be created as its own object. It handles the ticking of flame and vomit and removes them from the game after the set number of ticks (2).

class Flame : public Projectile

{

virtual void doSomething();

class Vomit : public Projectile

{

virtual void doSomething();

Flame and Vomit behave similarly. Both their doSomething functions check the entire list of actors to see if they can be afflicted by either flame or vomit. They handle changing the states of other actors when applicable (ie. when Penelope is infected, or when a goodie is destroyed by flame)

class Citizen : public Infectable

{

virtual void doSomething();

//does all of citizens prescribed behavior in the spec. described further in the pseudocode section

};

class Zombie : public Mortal

{

inline int getMovementPlan() const { return m\_movementPlan; }

void setMovementPlan(int newPlan) { m\_movementPlan = newPlan; }

void decMovementPlan() { m\_movementPlan--; }

};

I created a Zombie class to handle most of DumbZombie and SmartZombie’s shared functionality, namely the movement plan. Both of these inherit from the Zombie class.

class DumbZombie : public Zombie

{

virtual bool canHoldVaccine() const { return true; }

//only actor that can hold vaccine

virtual void doSomething();

//moves around randomly and when applicable vomits

};

class SmartZombie : public Zombie

{

virtual void doSomething();

//behavior as prescribed by the spec. explained further in the pseudocode

# **Pseudocode for Complex Functions**

int StudentWorld::init()

{

create starting conditions of the level

load level from level file

repeatedly

check if char in level file and translate it to game pixels

tell game to continue

}

int StudentWorld::move()

{

display scoreboard (check for negative score too)

give player chance to do something

repeatedly

go through each actor and give it chance to do something if its alive

remove dead actors

check if level finished or player died

tell game to continue

}

void StudentWorld::cleanUp()

{

check if last level was loaded (to avoid cleaning up something twice)

loop through all actors and remove if dead

}

void StudentWorld::createFlame(Actor\* caller)

{

check if caller was player or landmine

if player

for each of three flames, check if creation is permitted (ie. not blocked)

if not, create flame in the same direction as player

if landmine

create flame at calling location, as well as all 8 adjacent squares

create pit at calling location

}

void Penelope::doSomething()

{

check if not alive

check if infected, check infection timer

get user input

if directional keys, check movement block then move Penelope

if space, call createFlame after checking inventory

if enter, reset infection timer after checking inventory

if tab, drop a landmine

}

void DumbZombie::doSomething()

{

check if alive

check if paralyzed

check if zombie should vomit

check preexisting movement plan

choose random direction

move to location if not blocked

update movement plan

}

void SmartZombie::doSomething()

{

check if alive

check if paralyzed

check if should vomit

check preexisting movement plan

if nearby human further than 80 pixels away, move in random direction

otherwise choose randomly between directions that get zombie closer to human

move to location if not blocked

update movement plan

}

void Citizen::doSomething()

{

check if alive

check if paralyzed

check if closer to Penelope than zombie (within 80 pixels)

if so, pick between directions that get citizen closer to Penelope

move to new location

if not, calculate distances from nearest zombie after moving either up, down left or right

pick furthest distance and move there

if distances are less than original zombie distance, stay still

}

# **Design Assumptions**

When there’s an equal distance between the nearest zombie and Penelope, I implemented the citizen class such that it tries to move away from the zombie before moving closer to Penelope. In zombie’s doSomething(), the spec says to check for if its died from a flame. I handled this in the flame class instead. When calculating potential locations to move away from zombies, assuming there were multiple directions that led to equal distance away from the nearest zombie, I set the direction movement priority as such: up, down, left, right.

# **Class Testing**

I tested all my goodies in the same way. I made sure that when Penelope walked over them, her inventory and the scoreboard were incremented correctly. I also made sure that contact with flames destroyed them. In addition to that, I tested to make sure that contact with a citizen/zombie did not consume the goodie.

For my vaccine goodie, I did additional testing to make sure that dumb zombies were flinging vaccines randomly on death. I made sure that the vaccines they flinged could also be picked up, as well as destroyed by flames.

I tested my landmines by making sure they exploded when contacting Penelope, Zombies and Citizens. I also made sure that they could be set off with Penelope's flamethrower. In addition, I also tested chain reactions with land mines setting off other landmines. I also tested that the Pit object created where the landmine went off worked like a normal pit.

I tested my pits by making sure that they killed zombies, citizens and Penelope alike. I also checked to make sure that flames and vomit properly overlapped pits without being destroyed.

I tested my flames by making sure they killed Penelope, citizens, Zombies and goodies. I checked to make sure that walls and exit block the flames when the objects overlap (distance between centers is less than 10 pixels). I thought I found a bug because, when positioned correctly, the flames created by Penelope could pass through the middle of two adjacent walls/exits. However, I checked if this bug existed in the sample executable and, sure enough, it did. Therefore, I left it as is, because I didn’t want my program to function differently from the sample executable.

I tested walls by making sure Penelope, zombies and citizens couldn’t pass through them, and by making sure that flames and vaccines couldn’t be created inside or past them (except in the bug described above).

I tested exits by making sure that penelope, citizens and zombies could overlap them, and that penelope (when no citizens were left) and citizens could use them to exit the level.

I tested vomit by making sure that, when it overlapped with Penelope or citizens and they weren’t already infected, it made them infected. Furthermore, I tested that vomit didn’t infect any other actors that it overlapped (zombies, exits, pits).

I tested Penelope by making sure that she died when her infection count reached 500, or anytime she overlapped with a flame or a pit. I tested her movement by making sure she could move anywhere that wasn’t covered by a wall, zombie or citizen. I tested that, when she had the sufficient charges, she could produce flames, drop a landmine or heal herself (even if she wasn’t already infected). I also made sure that the above actions did nothing when she didn’t have the necessary charges.

I tested that all of my zombies and citizens were paralysed by comparing the speed with which they moved in my program to that in the sample executable.

Furthermore, I made sure that dumb zombies moved randomly no matter how close they were to a human, and vomited on a human only when they were facing them and were within range.

I made sure that smart zombies moved randomly when there was no human nearby, but moved in a random direction **towards** the nearest human when there was a human nearby.

I tested citizen by making sure that they didn’t move when zombies and Penelope weren’t nearby. Furthermore, when Penelope was closer to a citizen than any zombie, I made sure that citizen moved towards Penelope rather than away from the zombie, and vice versa when a zombie was closer than Penelope.

# **Bugs and Unfinished Functionality**

There was no unfinished functionality, but there were a few bugs:

* Citizens sometimes had to get one pixel closer than they were supposed to in order to use the exit
* When there are multiple citizens adjacent to each other, with a zombie nearby, sometimes the citizens would freeze until the zombie had been killed