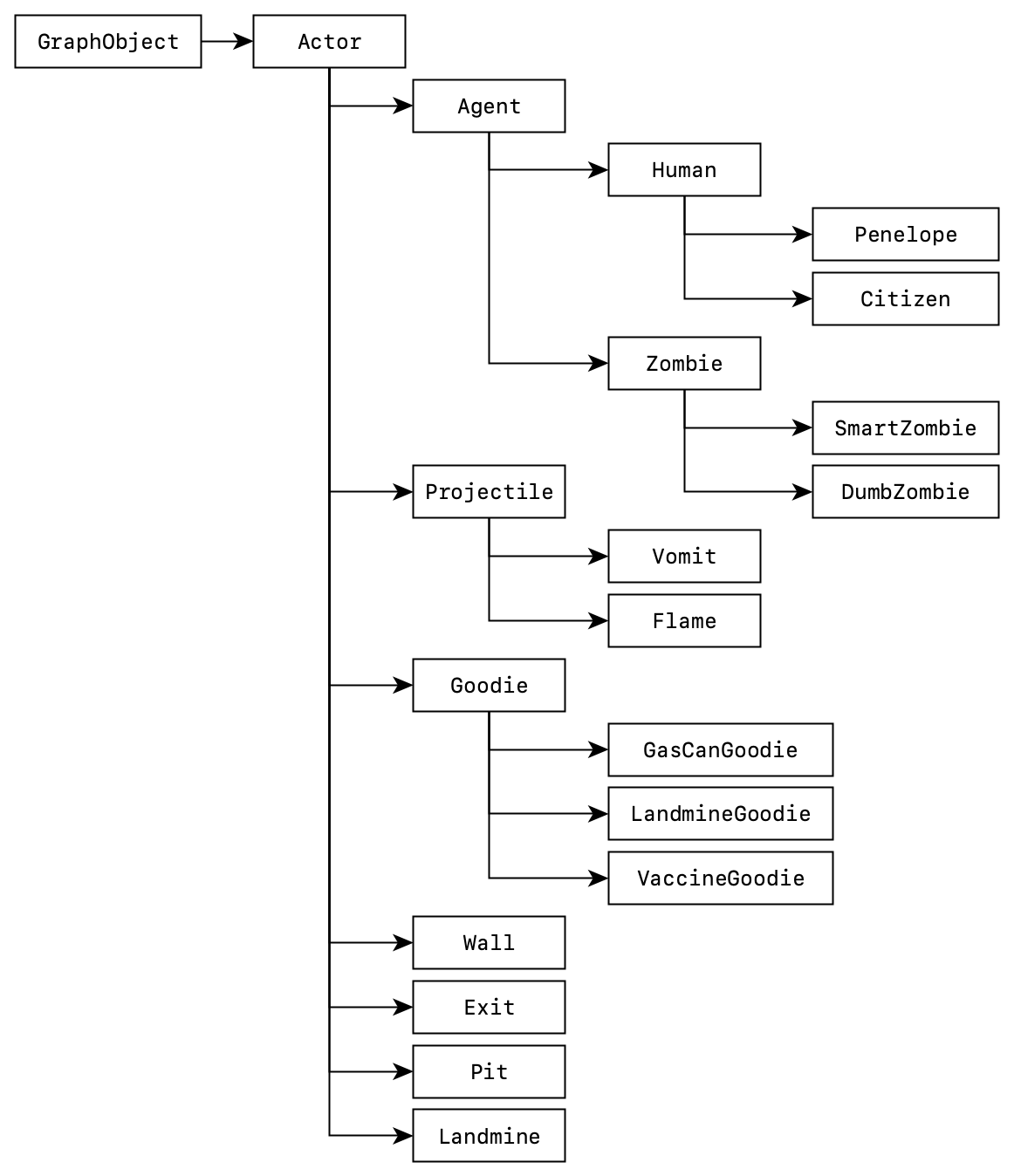
**Project 3 Report**

**Part 1. Overview**

My project 3 employs the following design to achieve the desired functionality. This diagram describes the class inheritance structure of the actor classes.

**This project is complete.**



In addition, the following macros are defined for clarity purposes.

|  |
| --- |
| #define OVERLAP\_DISTANCE 10  #define NEAR\_DISTANCE 80  #define RIGHT 0  #define UP 90  #define LEFT 180  #define DOWN 270  #define SCORE\_GOT\_GOODIE 50  #define SCORE\_CITIZEN\_DIE -1000  #define SCORE\_CITIZEN\_SAVED 500  #define SCORE\_DUMB\_ZOMBIE\_DIE 1000  #define SCORE\_SMART\_ZOMBIE\_DIE 2000  #define SPEED\_PENELOPE 4  #define SPEED\_CITIZEN 2  #define SPEED\_ZOMBIE 1  #define PROJECTILE\_LIFE 2  #define FLAMETHROWER\_RANGE 3  #define TIME\_INFECTION\_LIMIT 500  #define ZOMBIE\_MOVEMENT\_RANGE\_MIN 3  #define ZOMBIE\_MOVEMENT\_RANGE\_MAX 10 |

**Part 2. Class Designs**

**1) StudentWorld**

studentWorld oversees the general status of the game, and is also in charge of dealing with interactions between actors. In a sense, studentWorld acts as the god, orchestrating the whole game.

Below is the class declaration of StudentWorld. Since there are too many public member functions, and many of them are self-explanatory, I will choose some public members to elaborate a little on.

|  |
| --- |
| **class** StudentWorld : **public** GameWorld {  **public**:  StudentWorld(string assetPath);  **virtual** **int** init();  **virtual** **int** move();  **virtual** **void** cleanUp();  *// Accessors*  **int** humanCount() **const** { **return** m\_nHumans; }  **bool** levelFinished() **const** { **return** m\_levelFinished; }  **bool** playerDied() **const** { **return** player()->willBeRemoved(); }  Penelope \*player() **const** { **return** m\_player; }  **void** setPlayer(Penelope \*player) { m\_player = player; }  **void** addActor(Actor \*actor) { m\_actors.push\_back(actor); }  **void** removeActor(Actor \*actor) { m\_actors.remove(actor); }  *// Actor adders*  **bool** addPlayer(**const** **double** &x, **const** **double** &y);  **bool** addCitizen(**const** **double** &x, **const** **double** &y);  **bool** addSmartZombie(**const** **double** &x, **const** **double** &y);  **bool** addDumbZombie(**const** **double** &x, **const** **double** &y);  **bool** addVaccineGoodie(**const** **double** &x, **const** **double** &y);  **bool** addGasCanGoodie(**const** **double** &x, **const** **double** &y);  **bool** addLandmineGoodie(**const** **double** &x, **const** **double** &y);  **bool** addExit(**const** **double** &x, **const** **double** &y);  **bool** addWall(**const** **double** &x, **const** **double** &y);  **bool** addPit(**const** **double** &x, **const** **double** &y);  **bool** addLandmine(**const** **double** &x, **const** **double** &y);  **bool** addFlame(**const** **double** &x, **const** **double** &y, **const** Direction &dir);  **bool** addVomit(**const** **double** &x, **const** **double** &y, **const** Direction &dir);  *// Helpers*  **bool** squareOverlap(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2, **const** **double** &sideX,  **const** **double** &sideY) **const**;  **bool** squareOverlap(**const** Actor \*obj, **const** **double** &x2, **const** **double** &y2,  **const** **double** &width, **const** **double** &height) **const**;  **bool** squareOverlap(**const** Actor \*obj1, **const** Actor \*obj2, **const** **double** &width,  **const** **double** &height) **const**;  **int** distBetween(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2) **const**;  **int** distBetween(**const** Actor \*obj, **const** **double** &x, **const** **double** &y) **const**;  **int** distBetween(**const** Actor \*obj1, **const** Actor \*obj2) **const**;  **bool** checkOverlap(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2) **const**;  **bool** checkOverlap(**const** Actor \*obj, **const** **double** &x, **const** **double** &y) **const**;  **bool** checkOverlap(**const** Actor \*obj1, **const** Actor \*obj2) **const**;  **bool** checkNear(**const** Actor \*obj1, **const** Actor \*obj2) **const** {  **return** distBetween(obj1, obj2) <= NEAR\_DISTANCE;  }  *// Actions*  **void** infectAt(**const** **double** &x, **const** **double** &y);  **void** evacuateAt(**const** **double** &x, **const** **double** &y);  **void** killAt(**const** **double** &x, **const** **double** &y);  *// Predicates*  **bool** checkFlammableAt(**const** **double** &x, **const** **double** &y);  **bool** checkBlockAtDir(**const** Actor \*obj, **const** Direction &dir,  **const** **double** &dist) **const**;  **bool** checkOverlapWithHuman(**const** **double** &x, **const** **double** &y) **const**;  **bool** checkOverlapWithAgent(**const** **double** &x, **const** **double** &y) **const**;  Actor \*getNearestZombie(**const** **double** &x, **const** **double** &y) **const**;  Actor \*getNearestHuman(**const** **double** &x, **const** **double** &y) **const**;  *// Random*  **int** unifRandomInt(**const** **int** &min, **const** **int** &max) **const**;  **bool** bernoulliRandomBool(**const** **double** p) **const**;  Direction randomDirection() **const** { **return** unifRandomInt(0, 3) \* 90; };  **void** incHumanCount() { m\_nHumans++; }  **void** decHumanCount() { m\_nHumans--; }  **void** finishLevel() {  m\_levelFinished = **true**;  playSound(SOUND\_LEVEL\_FINISHED);  }  **private**:  list<Actor \*> m\_actors;  Penelope \*m\_player;  **int** m\_nHumans = 0; *// actors that need to be saved, Penelope included*  **bool** m\_levelFinished = **false**;  }; |

**Constructor** of Studentworld is trivial, hence I will skip it.

**lifecycle functions** are inherited from Gameworld. The lifecycle three functions controls the lifecycle of a game level.

|  |
| --- |
| **virtual** **int** init();  **virtual** **int** move();  **virtual** **void** cleanUp(); |

init() is called only once at the beginning of the level's lifecycle. It handles the initialization of a level, which includes resetting private variables of the level, constructing the level map and actors. It also decides whether the player has finished all the levels and therefore has won the game.

move() is called every tick after the level is initialized. It updates the GameStatText and controls the status of the game. More importantly, it gives all the actors a chance to do something, and then help deal with the consequences of their actions.

cleanup() is called once the level is ended for whatever reason. It deletes all the actors that are held in the m\_actors, and clean up the pointers.

**Accessor functions**, as their name indicates, provide read-only access to some private member variables or their variant. It also helps to simplify the logic such that it is straightforward. Without a doubt, they are const functions.

|  |
| --- |
| **int** humanCount() **const** { **return** m\_nHumans; }  **bool** levelFinished() **const** { **return** m\_levelFinished; }  **bool** playerDied() **const** { **return** player()->willBeRemoved(); }  Penelope \*player() **const** { **return** m\_player; } |

Take playerDied() for example, it helps to summarize a predicate that would change the game's status, which makes the following code block possible.

|  |
| --- |
| // StudentWorld.cpp  **if** (playerDied())  **return** GWSTATUS\_PLAYER\_DIED; |

**Actor adder functions**, for each non-pure-virtual actor, an actor adder function is provided.

|  |
| --- |
| **bool** addPlayer(**const** **double** &x, **const** **double** &y);  **bool** addCitizen(**const** **double** &x, **const** **double** &y);  **bool** addSmartZombie(**const** **double** &x, **const** **double** &y);  **bool** addDumbZombie(**const** **double** &x, **const** **double** &y);  **bool** addVaccineGoodie(**const** **double** &x, **const** **double** &y);  **bool** addGasCanGoodie(**const** **double** &x, **const** **double** &y);  **bool** addLandmineGoodie(**const** **double** &x, **const** **double** &y);  **bool** addExit(**const** **double** &x, **const** **double** &y);  **bool** addWall(**const** **double** &x, **const** **double** &y);  **bool** addPit(**const** **double** &x, **const** **double** &y);  **bool** addLandmine(**const** **double** &x, **const** **double** &y);  **bool** addFlame(**const** **double** &x, **const** **double** &y, **const** Direction &dir);  **bool** addVomit(**const** **double** &x, **const** **double** &y, **const** Direction &dir); |

Each actor adder function deals with one simple task, add the actor with given parameters if possible. If the actor, for whatever reason, cannot be added, the function returns false.

Take addFlame() for example. If a flame object cannot be created at the given coordinate, the function would return false refuse to add a flame. This is very handy when writing the Penelope::useGasCan() function.

These functions I write satisfy the requirements of the spec. However, it can be improved. For example, the addWall function does not really check anything; it simply adds a wall and returns true, since the spec does not say whether in what condition a wall cannot be added. I decide to leave it as is.

**Helper functions** provide the studentWorld with insight on the metrical relationships between different actors and other facts.

|  |
| --- |
| **bool** squareOverlap(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2, **const** **double** &sideX,  **const** **double** &sideY) **const**;  **bool** squareOverlap(**const** Actor \*obj, **const** **double** &x2, **const** **double** &y2,  **const** **double** &width, **const** **double** &height) **const**;  **bool** squareOverlap(**const** Actor \*obj1, **const** Actor \*obj2, **const** **double** &width,  **const** **double** &height) **const**;  **int** distBetween(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2) **const**;  **int** distBetween(**const** Actor \*obj, **const** **double** &x, **const** **double** &y) **const**;  **int** distBetween(**const** Actor \*obj1, **const** Actor \*obj2) **const**;  **bool** checkOverlap(**const** **double** &x1, **const** **double** &y1, **const** **double** &x2,  **const** **double** &y2) **const**;  **bool** checkOverlap(**const** Actor \*obj, **const** **double** &x, **const** **double** &y) **const**;  **bool** checkOverlap(**const** Actor \*obj1, **const** Actor \*obj2) **const**;  **bool** checkOverlapWithHuman(**const** **double** &x, **const** **double** &y) **const**;  **bool** checkOverlapWithAgent(**const** **double** &x, **const** **double** &y) **const**;  **bool** checkNear(**const** Actor \*obj1, **const** Actor \*obj2) **const**;  **bool** checkFlammableAt(**const** **double** &x, **const** **double** &y) **const**;  **bool** checkBlockAtDir(**const** Actor \*obj, **const** Direction &dir,  **const** **double** &dist) **const**; |

For convenience, some of the functions are overloaded a couple times. Of course all of them are const functions, since they only observe and do not make changes.

**Action functions** can be imagined as the physical law of the world, they handles the consequences of actors' actions. As such, they have real impacts, and are not constant functions.

|  |
| --- |
| **void** infectAt(**const** **double** &x, **const** **double** &y);  **void** evacuateAt(**const** **double** &x, **const** **double** &y);  **void** killAt(**const** **double** &x, **const** **double** &y); |

Each of the functions calls the corresponding actors' public function, if the actors (who are still in the game) overlaps with the coordinate.

infectAt: Actor::infect()

evacuateAt: Actor::evacuate()

killAt: Actor::die()

**Random generators** are simple encapsulations of C++ <random> library's random generators. Since they do not change anything, but merely provide random numbers. They are constant functions.

|  |
| --- |
| **int** unifRandomInt(**const** **int** &min, **const** **int** &max) **const**;  **bool** bernoulliRandomBool(**const** **double** &p) **const**;  Direction randomDirection() **const**; |

**Modifier functions** really do not have much to talk about. they help modify private member variables.

|  |
| --- |
| **void** incHumanCount();  **void** decHumanCount();  **void** finishLevel(); |

**2) Actor**

Actor is the base class of all other actor classes (duh), and is derived from the GraphObject class, which provides the interface with the GameWorld. Given the restriction that StudentWorld can only hold a single list of Actor objects, Actor need to define the interface that StudentWorld can interact with.

**Constructor** of Actor, in addition to what GraphObject requires, takes in an extra parameter StudentWorld \* world such that it can have ask world questions or demand world actions.

**Action functions** are actions that can be invoked by StudentWorld. Other than doSomething(), 3 other actions are defined, die(), infect(), and evacuate(). Since some of the derived actor classes have different behavior when these function are called, these functions are virtual.

However, these functions are not pure-virtual for some reasons I will talk about next.

|  |
| --- |
| **virtual** **void** doSomething() {  m\_turnFinished = **false**;  **if** (willBeRemoved())  finishTurn();  }  **virtual** **void** die(){}  **virtual** **void** infect(){}  **virtual** **void** evacuate(){} |

For doSomething(), although each derived actor class has different behaviors, they usually share a common part with their parent class. Hence, a cascading mechanism is introduced here to help avoid redundant code.

Namely, a private member variable m\_turnFinished is used to indicate whether the actor has finished doing something, and will no longer make any move during this tick.

Hence, the base class Actor need to reset the m\_turnFinished status at the beginning of each doSomething() call.

It also checks if the given actor is dead (or in my program, is set to be removed), if the actor is already off the game, it should not do anything and should immediately finish turn.

For die(), infect(), and evacuate(), because most of the classes have similar behaviors for these functions, which is do nothing, an empty default implementations are provided for each action.

**Accessor functions** either provide access to private member variables, or describes some properties that the Actor possess. The property-describing accessors are virtual functions with default implementations.

|  |
| --- |
| StudentWorld \* world() **const** { **return** m\_world; }  Penelope \* player() **const**;  **virtual** **bool** isHuman() **const** { **return** **false**; }  **virtual** **bool** isZombie() **const** { **return** **false**; }  **virtual** **bool** blocks() **const** { **return** **false**; }  **virtual** **bool** isFlammable() **const** { **return** **true**; }  **bool** willBeRemoved() **const** { **return** m\_willBeRemoved; }  **bool** turnFinished() **const** { **return** m\_turnFinished; } |

**Helper functions** provide metrics information, either relative to some other actor object, or relative to itself. Since they do not change anything, they are all const functions.

|  |
| --- |
| **double** overlaps(**const** Actor \*obj) **const**;  **double** distTo(**const** Actor \*obj) **const**;  **bool** isNear(**const** Actor \*obj) **const**;  **double** offsetX(**const** Direction &dir, **const** **double** &dist) **const**;  **double** offsetY(**const** Direction &dir, **const** **double** &dist) **const**; |

finishTurn() was introduced above when we talk about doSomething() which set tells the doSomething() function to do nothing more.

|  |
| --- |
| **void** finishTurn() { m\_turnFinished = **true**; } |

scheduleRemoval(), is essentially "setDead". Once an actor is scheduled to be removed, it will no longer do anything, and wait to be deleted at the end of the tick.

|  |
| --- |
| **virtual** **void** scheduleRemoval() {  finishTurn();  m\_willBeRemoved = **true**;  } |

**3) Agent**

Agent is a base class for actors that can move around on their own, namely, zombies (smart and dumb), and humans (penelope and citizens).

The agents all have some speed, all except penelope paralyze every other turn. Hence, agents have two private member variables m\_speed and m\_isParalyzed, and can be accessed by speed() and isParalyzed(). Also, because they block the movement of each other, their block() function returns true.

Agent also have two helper functions:

|  |
| --- |
| bool isBlockedAtDir(**const** Direction &dir) **const** |

This function checks if the agent is blocked at the given direction, meaning whether the agent can move towards that direction by distance speed().

|  |
| --- |
| Direction getDirectionTowards(**const** Actor \*target) **const** |

This function get the direction of the target actor relative this agent, its implemented as follow:

1. If the target is at the same row or same col, then return the direction that faces the target directly.

2. Otherwise, choose the composition of the real direction

(e.g. target at northeast, then choose north (up) and east (right))

3. If one of the two directions are blocked, return the other direction.

4. If both are blocked or both are not blocked, return either with equal chance.

This function is very helpful in the implementation of citizens and smart zombies.

**4) Human**

Agent is a base class for agents that can be infected -- penelope and citizens. Since these two actor classes are both human, then isHuman() should return true (duhh).

Humans can be infected, so the following public functions

|  |
| --- |
| **virtual** **void** infect();  **void** updateInfection();  **bool** isInfected() **const**;  **int** infectionTime() **const**;  **virtual** **void** cure();  **virtual** **void** mutate() = 0; |

and

|  |
| --- |
| **private:**  int m\_infectedTime = -1; |

are added accordingly.

If m\_infectedTime = -1, the human is not infected.

If m\_infectedTime ≥ 0 , the human is infected for m\_infectedTime seconds.

infect() and cure() are a pair of functions that have opposite effects. When infect() is called, m\_infectedTime is set to 0, and the counting starts. On the contrary, cure() sets m\_infectedTime to -1.

mutate() is a pure virtual function, since when m\_infectedTime reaches 500, different things happen to citizen and penelope.

updateInfection() not only increments the m\_infectedTime, it also calls mutate() when infected time reaches 500.

|  |
| --- |
| **virtual** **void** doSomething(); |

Since all human, if infected, updates their infection in the same way, human's doSomething() function handles that, and also calls mutate once the limit is reached.

**5) Penelope**

Penelope is our little hero, and the actor that the player control. It has overrides 4 virtual actions functions, and have another 9 public functions related to 3 different types of goodies.

|  |
| --- |
| **virtual** **void** doSomething();  **virtual** **void** mutate();  **virtual** **void** die();  **virtual** **void** evacuate(); |

Penelope's doSomething(), in addition to human's doSomething(), need to respond to player's keystroke, and either move or use a type of goodie.

With the Agent's helper functions, move can be concluded in 3 lines:

|  |
| --- |
| setDirection(dir);  **if** (!isBlockedAtDir(dir))  moveTowards(dir); |

When a keystroke requires Penelope to use goodies, a use[Goodie]() function will be called, which I will discuss later.

Penelope's mutate() is equivalent to die(). Hence, it simply calls die().

When Penelope die(), the whole game ends. In my implementation, StudentWorld is informed of Penelope's death in no special way -- by checking if it is set to be removed.

When Penelope try to evacuate(), Penelope will check if every citizen has been saved using world->citizenCount() == 0. If so, level finished is signaled to studentWorld by world()->finishLevel().

For each goodie, pickup[Goodie](),[goodie]Count() and use[Goodie]() functions are implemented.

|  |
| --- |
| **int** gasCanCount() **const**;  **int** landmineCount() **const**;  **int** vaccineCount() **const**;  **void** pickupGoodie();  **void** pickupGasCan();  **void** pickupLandmine();  **void** pickupVaccine();  **void** useGasCan();  **void** useLandmine();  **void** useVaccine(); |

[goodie]Count() are private member variable accessors, nothing special.

pickupGoodie() is the common part of pickup[Goodie](). Essentially, it increases the player's score and play a sound. Each pickup[Goodie]() then increment the goodie count by the designed amount.

use[Goodie]() checks if the player has at least one goodie, decrease the amount of goodie by 1. Then do what the goodie is supposed to do. I am not going to walk through the details, since they are specified in the spec.

**6) Citizen**

Citizen, much like Penelope also has many of its unique actions. Hence 5 virtual action functions are overridden:

|  |
| --- |
| **virtual** **void** doSomething();  **virtual** **void** die();  **virtual** **void** evacuate();  **virtual** **void** mutate();  **virtual** **void** infect(); |

Citizen doSomething() follows a complex routine.

In addition to what a human does, (which includes mutate into a zombie), it paralyzes every other second. One thing that is involved in this process is that it need to know where the current zombie is. This can only be told by studentWorld using the nearestZombie() funcion. Most of the hard work are done by Agent's helper functions.

Citizen's die() and evacuate() handles the score and sound, and set citizen to be removed.

Citizen's mutate() unlike Penelope's, which is just die(), will turn the citizen into a zombie.

Citizen's infect(), in addition to what human's infect() does, also plays a sound.

**7) Zombie**

Zombies are not really that different, be it smart or dumb, they both vomit. The only difference is that, smart zombie moves in style, while dumb zombies drops goodies. (Also they have different scores.) With that observation, I put most of Zombie's implementation in the Zombie base class.

|  |
| --- |
| Zombie(StudentWorld \*world, **double** x, **double** y);  **bool** vomit();  **virtual** **void** doSomething();  **virtual** **void** die();  **virtual** **bool** isZombie() **const** { **return** **true**; }  **int** getPlan() **const** { **return** m\_plan; }  **void** setPlan(**const** **int** &plan) { m\_plan = plan; }  **void** decPlan() { m\_plan--; } |

getPlan(), setPlan(), decPlan() are used to modify m\_plan , which is the remaining movement plan steps.

doSomething() handles paralyzing as well as vomit. This will be called in both dumb and smart zombies.

There's really not much to talk about in die() -- it plays a miserable howling sound (is that you... Carey?).

**8) Smart Zombie**

SmartZombies in addition to what zombie does, also knows how to follow a nearby player. To do so, it needs guidance from StudentWorld::getNearestHuman() function.

**9) Dumb Zombie**

DumbZombie moves in freestyle, so it only need the random function from world. Other than that, it drops vaccine when it dies.

**10) Goodies (GasCanGoodie, LandmineGoodie, VaccineGoodie)**

Goodies, despite having exciting names, have pretty boring implementations. Given that what they do are essentially the same, I will talk about all of them together.

|  |
| --- |
| **virtual** **void** doSomething();  **virtual** **void** die() { scheduleRemoval(); } |

Since they can be burned, they can be destroyed. Hence, when die() they will be removed.

In doSomething() it checks if it overlaps with the player: if it does, the player gets the goodie pickup[Goodie]().

And that is the story of Goodies.

**11) Projectile**

Projectiles all have a life of 2 ticks. While they can, they do their worst.

The projectile base class only implements doSomething() where it counts down its life using m\_age, which is initially 0. Once m\_age reaches 2, it immediately scheduleRemoval() and do nothing else.

|  |
| --- |
| **virtual** **void** doSomething(); |

**12) Flame**

Flame in addition to what Projectile do, ask the world to kill at its location. It's whole doSomething() can be seen as follows:

|  |
| --- |
| **void** Flame::doSomething() {  Projectile::doSomething();  **if** (!turnFinished())  world()->killAt(getX(), getY()); // brutal!  } |

**13) Vomit**

Vomit in addition to what Projectile do, ask the world to infect at its location. It's whole doSomething() can be seen as follows:

|  |
| --- |
| **void** Vomit::doSomething() {  Projectile::doSomething();  **if** (!turnFinished())  world()->infectAt(getX(), getY()); // ewww!  } |

**14) Landmine**

Landmine is a vicious creature. After, 30 ticks of inactive phase it will explode() if:

1. it is touched

2. it is killed

The explosion results in 9 flames, around and at its coordinate and leaves behind a pit.

|  |
| --- |
| Landmine(StudentWorld \*world, **double** x, **double** y);  **virtual** **void** doSomething();  **virtual** **void** die();  **void** explode(); |

**15) Pit**

Pit works like an eternal Flame. since it cannot be destroyed, scheduleRemoval() is overridden to be an empty function.

**16) Wall**

Wall... It blocks, it doesn't burn, and it can't be destroyed.

|  |
| --- |
| **virtual** **bool** blocks() **const** { **return** **true**; }  **virtual** **bool** isFlammable() **const** { **return** **false**; }  **virtual** **void** scheduleRemoval() {} |

**16) Exit**

|  |
| --- |
| **virtual** **void** doSomething();  **virtual** **bool** isFlammable() **const** { **return** **false**; }  **virtual** **void** scheduleRemoval() {} |

Exit, cannot burn, cannot destroy. Most importantly, it can evacuate the humans.

In fact, Exit also is a very simple class. Its doSomething function, in its entirety, is as follows.

|  |
| --- |
| **void** Exit::doSomething() {  *// I AM THE DOOR. If anyone enters by Me, he will be saved.*  *// (John 10:9-16 NKJV)*  Actor::doSomething();  **if** (!turnFinished()) {  world()->evacuateAt(getX(), getY());  }  } |

**Design assumptions**

The spec did not say that, when a dumb zombie drops a vaccine goodie and if the vaccine goodie is flinged into a wall (I mean overlap), what should the vaccineGoodie do. Well, I didn't do anything about it and just let it stuck in there.

**Part 2. Tests**

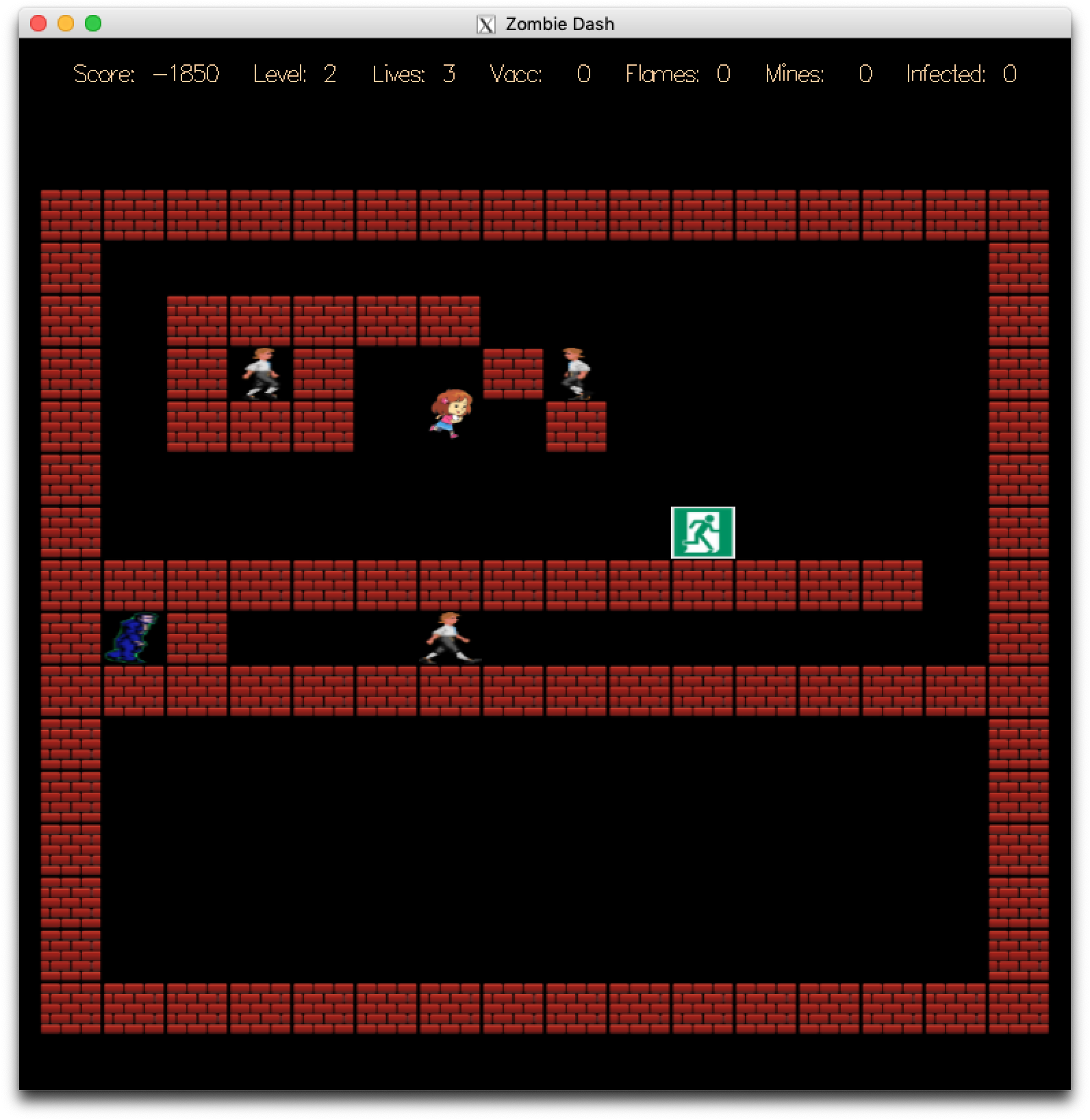
To test each of the class, I built 30 levels: 10 levels where Penelope get to interact with one single actor from each most derived class; and another 20 levels to test the interaction between the different actors.

**1) Citizen**

To test the behavior of the citizen, I build different types of cells to check how, per each movement scenario specified, will the citizen behave.

Then I build a cell with only zombie and the citizen, to see if the citizen can be correctly converted into a zombie.

Then I tested the citizen's follow pattern: given that a celled zombie is around, how will the citizen behave.



**1) Citizen**

To test the behavior of the citizen, I build different types of cells to check how, per each movement scenario specified, will the citizen behave.

Then I build a cell with only zombie and the citizen, to see if the citizen can be correctly converted into a zombie.

Then I tested the citizen's follow pattern: given that a celled zombie is around, how will the citizen behave.

**2) Zombies and vomit**

To test how robust the zombie behaves, I essentially filled one level all with smart zombies, and another level all with dumb zombies.

I want to test both the following pattern of the zombie, and in addition, whether the dumb zombies drop goodies correctly.

I also tried to get infected. I died, as I intended. Then I tried the vaccine, it works.

**3) Goodies, flame, and landmine**

I test the goodies by themselves, make sure picking up goodie behaves correctly.

Then, I tried the landmine goodie on zombies as well as citizens, to make sure that the triggering mechanism is correct. I then tested the flame on walls, exits, and pits, to make sure the correct things are burnable.

**4) Pit**

I led zombies and citizens into the pit, then jumped into the pit myself. It worked. (DEAD serious, no pun intended)

**5) Exit**

I led all the citizens into the exit first; then I went in, it worked.

I killed all the citizens into the exit first; then I went in, it worked.

I killed some citizens and then led some citizens into the exit first; then I went in, it also worked.

(Well the exit doesn't judge it seems)

**7) Checking sound**

I make sure all the sounds are played correctly by going through the master list on page 9 of the specs.

**8) Checking score modification**

I make sure all the scores are recorded correctly by going through the master list on page 6 of the specs.