1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Zombie Dash are able to sneeze, and each type of actor sneezes in a different way.”

**Actor:**

Note: I declared these functions in the Actor class so that this functionality would be accessible when I traverse my complete vector of Actor pointers. As a result, some of these functions extend to classes who have no use for them. These are also primarily

ways for the StudentWorld to identify certain qualities of the Actor pointer, such that it may act accordingly despite polymorphism. Only doSomething is pure virtual, because the other functions either shared common behavior between derived classes, or I slapped on a definition so a derived class wouldn’t have to later on.

virtual bool alive() const:Every Actor is alive when initialized, so this function serves as an accessor for StudentWorld::move() to know when to call that actor’s doSomething function and when to remove the actor from the game.

virtual void doSomething() = 0 : This function is also pure virtual in the GraphObject class, so its inclusion in the Actor class is more of a formality. Every Actor has a different procedure when doSomething is called, so I kept it pure virtual.

virtual bool isTangible() const: This boolean function helps the StudentWorld identify which actors (Walls and Agents) should not have overlapping bounding boxes and which ones are permissible to step on (Hazards, Exits, and Goodies). This function is set to return true within the Actor class, and Hazards, Exits, and Goodies have their own override to isTangible() to return false.

virtual bool canBeSaved() const: This boolean function primarily pertains to determining exit overlap, and is only true for the Person class and its derivations. As a result, the StudentWorld knows not to allow Zombies or miscellaneous landmines to enter the exit.

virtual bool isKillable() const: This boolean function informs the StudentWorld if the object may be damaged by the Flame class. I set it to false inside the Actor class, and overrode its definition to return true for the Goodie and Agent classes.

virtual bool zombieFood() const: This boolean function informs the StudentWorld if the object attracts Zombie attention when determining Zombie movement. This function returns false in the Actor class, and returns true for the Person class. It is a rather niche function, so its effectiveness is doubtful, but it was included for lack of time to think of a better workaround.

virtual void activateByFlame(): Its primary use is to notify the Landmine object if a Flame has touched it, and has no instructions for the other classes. Landmine::activateByFlame() instructs the landmine to explode.

virtual bool scaresCitizens(): This notifies the StudentWorld when it encounters Actors (i.e. Zombies) in the vector that the Citizen class should choose to avoid. This function is set to return true in the Zombie class, however returns false in the Actor class.

virtual void die(): When called, this function instructs an Actor to set its status to dead, since the private m\_alive variable resides in the Actor class. In the Zombie and Citizen classes, this function instructs the StudentWorld to play the appropriate sounds and change game score to reflect the change in events.

virtual int beSaved(): This function notifies Actors that are able to leave the level through the Exit that they have left. This action applies only to objects of the Person class (Penelope and Citizens).

virtual bool canBlockFlames() const: This function notifies the StudentWorld object whether an Actor forbids flames from appearing within its overlap range. At this point, it pertains true to only Walls and Exits.

virtual bool triggersLandmines(): This function notifies the StudentWorld whether an Actor will cause a landmine to explode if it overlaps with it. This statement is only true for Agents, which is why I made it virtual.

void getInfected(): This function is only called by the StudentWorld after zombieFood() returns true. It instructs the Actor object to set its infection status to true, which only applies to objects belonging to the Person class hierarchy.

**Agent:**

Move(Direction dir)

I used a switch statement to check the direction parameter and used the StudentWorld’s functions to check if a movement in that direction for a certain number of spaces was possible without having bounding problems. If such a movement is possible, the function calls GraphObject’s moveTo function to update the Agent’s location in the game. Move() returns true if the Agent was successfully moved, false otherwise. I put move() in the Agent class because its four ingame derived classes: Citizen, DumbZombie, SmartZombie, and Penelope all share the need to check the StudentWorld before moving and then moving a certain distance.

**Person:**

Person::getInfectionCount() is an accessor function that returns how many ticks a Person has been infected for. This function is useful to determine when a Citizen will turn Zombie and to update the game display of Penelope’s infection status. Since both derived classes deal with being infected, it makes sense that getInfectionCount() belongs here.

**Penelope:**

Penelope::plusFlame() increments Penelope’s flame charge inventory by 5, and is called when Penelope interacts with a gas can goodie. Since only Penelope has flamethrower functionality, plusFlame() belongs here.

Penelope::getFlame() returns Penelope’s flame charge inventory count, and is primarily used to update the game display.

Penelope::getMines() returns how many mines Penelope has in her inventory, and is also used to update the game display.

Penelope::getVaccine() returns how many Vaccines Penelope has in her inventory, and is used to update the game display.

Penelope::plusMines() increments Penelope’s mine count by 2, and is called when Penelope interacts with a landmine goodie.

Penelope::plusVaccines() is called when Penelope interacts with a vaccine goodie, and increments Penelope’s vaccine count by one.

Penelope::doSomething()

I wrote Penelope’s doSomething with a series of switch statements dependent on what key the player pressed. Her movements were taken care of by Agent::move(), and I used public functions from StudentWorld to add Flame and Landmine objects into the game as needed.

The only change with Penelope::die() is that it plays SOUND\_PLAYER\_DIE, then calls Actor::die()

**Citizen:**

Citizen::doSomething()

I wrote Citizen’s doSomething with a structure of if statements to check off whether he turns into a zombie or is paralyzed, then I calculated his potential movements within the StudentWorld class.

**Zombie:**

I wrote Zombie::vomit using a structure of if statements to represent the Zombie’s decision making process, and I called randInt to determine whether or not the Zombie will instruct the StudentWorld to make a new Vaccine object. The function is not virtual because DumbZombie and SmartZombie use the same algorithm, so a call to Zombie::vomit will be enough.

Zombie::die() instructs the StudentWorld to play the appropriate zombie dying sound, then calls Actor::die()

**DumbZombie:**

DumbZombie::die()

I calculated whether or not the zombie will throw a vaccine at a random space around with two calls to randInt, then called Zombie::die().

I wrote DumbZombie’s doSomething in a series of if statements, checking if it was either paralyzed or it used its turn to vomit. randInt was helpful in determining a new movement plan. Then I called Agent::move() to see whether or not I had to reset the movement plan with a failure from move().

**SmartZombie:**

I wrote SmartZombie’s doSomething in a series of if statements, checking if it was either paralyzed or it used its turn to vomit.To calculate a movement plan, randInt worked similar to its use in DumbZombie, then I moved all the math to StudentWorld::goAfterHumanIfApplicable. If the zombie wasn’t close enough to a human, then I called Agent::move() to move it along its the current direction.

**Flame:**

I wrote Flame’s doSomething mostly in the StudentWorld class because I had to check for overlap under hazardOverlap() and for every flammable actor in range, set dead. The only functionality doSomething itself has is that it determines if the flame’s two ticks are up and it should set its state to not alive.

**Landmine:**

Similar to Flame, I wrote the algorithm for overlap in StudentWorld because it involves multiple actors outside the landmine itself. Its doSomething mainly concerns itself with figuring out whether the activation period has passed (30 ticks) and then setting its activation status to true.

**Pit:**

A pit just sits there, unless something overlaps with it, then it calls the same overlap function as Flame.

**Exit:**

The doSomething for Exit doesn’t do much except call checkExit in StudentWorld to evaluate its surroundings for overlap.

**Vomit:**

At this point, I essentially wrote anything involving overlap in StudentWorld, so Vomit’s doSomething sets its state to dead after 2 ticks, and then calls StudentWorld’s vomitOverlap function.

**Goodies:**

The doSomething functions for Gas Can Goodie, Landmine Goodie, and Vaccine goodie all call StudentWorld’s goodieOverlap, and if it returns true, add their respective treats to Penelope’s inventory.

**StudentWorld**

none of these functions are virtual (except the functions derived from GameWorld) because StudentWorld is a derived class, nothing inherits its behavior.

virtual int init()

I used the ostringstream object to help write an algorithm to cycle through the levels. Then I followed the format on the spec to load the level specified by the ostringstream string. I applied the necessary return statements if a file was not found or formatted incorrectly. Then I used the provided nested for loops and switch statement to initialize actors into the game and add them into my actor container.

virtual int move()

For every actor in my container of actors, move() calls their doSomething functions provided that they are alive. If Penelope dies after a call to doSomething or she beats the level, the level exits immediately thanks to a couple of if statements. Then I update my actor vector with new actors that were created as a consequence of an Actor’s doSomething and placed into a temporary vector. Then I empty the total Actor container of any dead Actors.

virtual void cleanUp();

For every actor in my unified Actor vector, I delete them, and then erase their values from the vector. Then I either decrease Penelope’s lives or increase the level count as appropriate

Penelope \* getPenelope()

I chose to write a getPenelope() so that the Goodie class could access StudentWorld’s penelope pointer to update her inventory.

bool canMoveTo(double end\_x, double end\_y, Actor \* character);

I decided to write a canMoveTo method to determine if an actor can move to a certain spot without intersecting bounding boxes with a tangible actor. (end\_x, end\_y) represents the destination location of the actor of interest, and character points to the same actor. The pointer is necessary to ensure that the actor does not block itself, and so I calculated the four corners of the character’s potential bounding box at that location, with left bottom corner (end\_x, end\_y). If any of those corners intersected the bounding box of a nearby tangible actor, canMoveTo will return false. If otherwise, then the function will return true. canMoveTo was a handy algorithm to have on hand when writing Agent::move(), and has lot of use in coding SmartZombie and Citizen.

void checkExit(double curx, double cury);

To determine if a citizen or Penelope has reached the exit, checkExit is an extension of Exit’s doSomething. It traverses the Actor container and uses StudentWorld’s protected overlap function and Actor::canBeSaved to determine if an Actor may exit the level.

All the overlap functions do roughly the same thing, except they address every actor that fulfills their conditions differently. Every overlap function traverses the actor vector and determines if each element overlaps with the actor at (curx, cury), and are called by their respective Actor types’ doSomething.

void hazardOverlap(double curx, double cury);

For every overlapping actor that is killable, they die. Used by flames and pits

bool goodieOverlap(double curx, double cury);

If an overlapping actor is Penelope, return true. Else, return false.

bool mineOverlap(double curx, double cury, Actor \* caller);

If an overlapping actor can trigger a landmine, return true, else return false

void vomitOverlap(double curx, double cury, Actor \* caller);

For every overlapping actor that is infectable, they get infected

I also implemented a series of generate functions to address Penelope’s usage of flames and landmines, Zombie’s vomit, etc.

void generateVaccine(double x, double y);

This creates a vaccine at location (x,y).

void generateFlames(double x, double y, int dir);

A Flame located at (x,y) spawns with direction dir in response to Penelope’s use of the flamethrower.

void generateLandmine(double x, double y).

A landmine is spawned at (x,y).

void generatePit(double x, double y);

A Pit is spawned at (x,y) as a consequence of exploding landmine.

void generateVomit(double x, double y, int dir)

Vomit is spawned at (x,y) in the direction specified

void generateZombie(double x, double y);

A Zombie is spawned at location (x,y) as a consequence of infection

void moveCitizen(Agent \* thisGuy);

I first calculated the Citizen’s distance from Penelope, then the distance from the nearest Zombie. Depending how those values compared, I followed the spec’s instructions on whether the Citizen should walk toward Penelope or run away from the Zombie. In the case that the Citizen walks towards Penelope, I used a combination of if statements to determine which direction and movement the Citizen should take. Otherwise, when the Citizen tries to avoid the Zombie, so for each direction the Citizen calculates four separate distances from zombies and chooses the direction that will bring him the farthest. If all four directions bring him closer, the Citizen stands still. If nothing provoked him in the first place, the Citizen stands still.

bool goAfterHumanIfApplicable(Agent \* zombie, double x, double y)

Similar to the Citizen’s algorithm for following Penelope, this is the SmartZombie’s attempt to get closer to Penelope or a citizen. If a citizen/Penelope is within a euclidian distance of 80, the zombie will turn towards them and follow. If not, the zombie’s doSomething will generate a random direction for it to follow.

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

This function is used to determine if a Citizen or Penelope is within vomiting range of a Zombie.

I chose to write a decCitizens function to help update StudentWorld’s private Citizen count so the Exit knows when to take in Penelope.

I decided to write a decZombies function to help update StudentWorld’s private zombie count variable for Citizen movement calculations.

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Flame class.” or “My smart zombie doesn’t work correctly yet so I treat it like a dumb zombie right now.”

I wasn’t able to really get the SmartZombie or Citizen to randomly choose between the two directions that would bring them closer to a Citizen/Penelope or Penelope respectively. The SmartZombie and the Citizen will always favor one direction over the other.

3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”

The spec allowed that new Actors initialized while StudentWorld::move() was running to wait until the next tick before move() called the new Actors’ doSomething(), so I made a secondary container to temporarily house these new objects.

4. A description of how you tested each of your classes (1-2 paragraphs per class).

Penelope was fairly simple to test, since the algorithm to move her was the player’s input. I tested her movement capacity by running her into various objects in levels 1 and 4, e.g. making sure she did not walk through walls or other people. I tested her other functionality by running her through goodies, making sure she picked them up, and she was able to use them. Since a few of her private data members were stored in her base classes, I had to make sure those were accessed properly. The game display above the level was a good way to gauge what worked and what didn’t. Other things to look out for were making sure that Penelope’s exit did not return the same result as her dying, she didn’t lock herself in her own bounding box, and ensuring that she moved the right number of ticks. At one point, I realized that her movement was bound to one space at a time when she was not able to outrun the landmine she placed, which was remedied soon after.

I initially tested the citizen by rewriting level five to have one citizen walking alone through a horde of dumb zombies to see how effective the citizen walked away from nearby zombies. It was easy to see if the basic functionality was present such as hearing the sound of citizens dying or entering the exit, and watching the game score change as a result. I noticed too that the citizens had a bad habit of clustering around Penelope, which was true to the example. In those cases, I noted that the flames successfully overlapped with the citizens when I threw a mine down. I used level 3 to test if a citizen would turn into a zombie after infection. Level 4 was a good setup to test citizens’ response to hazards, particularly the pit. The citizens did not walk through walls or other agents because of the algorithm I wrote for Penelope.

Dumb zombies were also simple to test since their movement was easy to compare to that of the sample zombies. Since the vomit algorithm involved overlap, it was straightforward to debug since it was similar to my other functions in StudentWorld. I used Penelope to test that they died correctly and that they infected others correctly. Level 3 allowed me to verify that they vomit correctly and that Citizens will turn into zombies after 500 ticks. Unlike Penelope, it was harder to move the dumb zombies around so they would trigger mines. The randomized vomit algorithm was observed by zombies walking toward citizens and Penelope, and success was easy to tell by the zombie vomit noises. Level 4 was a good testing ground to watch zombies fall through pits. After gathering a bunch of them in level 5, I proceeded to shoot them with fire until they started coughing up vaccines.

Smart Zombies were mainly tested by placing Penelope near a horde of them and watching them turn toward Penelope. Similar to their less intelligent counterparts, I checked if they fell through pits and died by flame correctly. They were easier to test the vomit functionality on as they were predisposed toward Citizens and Penelope. When they bumped together, often owed to Penelope’s change in direction, they visibly changed directions, indicating success in the movement plan. However, due to the flaw of having the same direction preferences as one another when moving toward a Citizen, their movement is not as visually spastic as in the sample program. Like other Agents, they did not walk through walls. They successfully walked onto the exit without triggering any event.

The goodies were very similar to each other in terms of testing. The main priority was to ensure Penelope could pick them up, and that they didn’t do anything in contact with other walking agents. After that, I tested to make sure flames removed them from the game field, and that vomit wouldn’t. I placed mines on them to make sure that they didn’t trigger them. They didn’t visibly interfere with the zombie or citizens’ movement plan, so I determined I was all set on that side of things. When Penelope walked over them, they quickly vanished, and Penelope’s item count increased, so thus I concluded my testing of the goodies.

I evaluated the flame for its effectiveness in slaying zombies and people alike, and I tested that it triggered the landmine. I also made sure it removed goodies, but not other hazards like other flames and vomit. The flame successfully crossed the pit and exit, and didn’t block itself. The flame remained for roughly the same time on screen as the sample executable, and killed everything that could be killed. The wall successfully stopped the flame’s advance, and consumed a zombie’s thrown vaccine in the same direction.

The landmine would often blow up before Penelope could leave the area, but I determined it was due to the program not reading the key I kept held down. I’d have to repress the desired direction quickly before the landmine exploded. Otherwise, the landmine explodes upon contact with the zombie, the citizen, and Penelope. It safely sits on the exit, and sometimes doesn’t kill citizens when they attempt escape because exit::doSomething() gets called first. It does not block the advancing flame, and the blast radius displays without a hitch, killing everything. The best way to test the landmine is when Penelope is stuck in between a bunch of citizens with no escape. It was hard to test whether vomit crosses the landmine successfully since the zombie wouldn’t vomit at that proximity from Penelope, and any further moves resulted in Penelope, the zombie or both blowing up.

The vomit object was more or less only applied toward Citizens, so whether it remains on walls or not remains obscure. Since it’s more or less impossible to have a zombie vomit at a citizen from across a wall, my program currently does not check if there is a wall before generating the vomit object. The vomit object remains for about two ticks and then cleans itself up. It successfully infects citizens and Penelope, and crosses exits and goodies. Vomit does not trigger landmines, although it is hard to verify as stated before. It is not killed by flame; pretty much the only life altering factor is the two tick timer.

The pit does not move from where it is originated. Level 4 is a good test of the pit’s functionality as the farthest north Citizen will have an easy time falling in. It does not impair movement for flame or vomit classes, and can exist on top of an exit without killing it. Penelope, Citizens, and both Zombies die upon contact. It is unlikely to test how a pit will affect the goodies, since there is no scenario that they would be in overlap range of each other. A pit can overlap with another pit without event. A pit cannot be killed by anything, which follows.

I tested the exit by leading a couple of zombies on top, noting that nothing happens. Then I put a landmine on the exit and led a couple of zombies onto the exit, noting that they triggered the landmine. Citizens enter the exit without a problem, and the sound plays. Exits do not impair the movement of flames and vomit, and I tested that they cannot be killed. Once the last citizen is either saved or zombified, the exit successfully accepts Penelope. The exit can overlap with the landmine without triggering it, and if the landmine explodes, the pit sits with the exit.

The walls are fairly simple to test. They do not allow anything to pass. Flames cannot overlap with it; agents can’t walk through them. Walls cannot be harmed by flame or infected by vomit. Walls cannot be killed. In terms of behavior, walls aren’t much to look at, they just sit there.

I tested the StudentWorld object by its ability to compile the game first off. The display needed to match that on the spec, and each object initialized on the level needs to be cleaned up. So I ran it on g++ to make sure there were no memory errors. Each object is initialized as the level map dictates it to be, and there are no mysteriously left over actors on the game map. The level is increased after each level finished and lives are lost every time Penelope dies. The interactions between Actors go mostly the way they should be.