1.

Student World functions:

**virtual** **int** init();

Sets up the actors in each level. Uses the level class to access the level files. The player “wins” after level 99, or if no other level file is found. With a correctly formatted level, the program loops through the level file, and if a marking is found indicating an actor, that actor is added to a list of Actors the StudentWorld class holds, known as gameObjects. After everything has been added, game continues. This function is virtual because it is present in the GameWorld class. This function is in StudentWorld because it pertains to the game as a whole, as opposed to a specific actor.

**virtual** **int** move();

Responsible for everything happening in a level. First, it loops through gameObjects, and for each Actor, their doSomething function is called. Then, Penelope’s doSomething is called. After that, it checks if Penelope has died; if so, a sound is played indicating that she died, and the function returns with the corresponding status. If Penelope can exit the game (now that all citizens have been saved), the canExit Boolean resets itself, the proper sounds are played, and the proper status is returned. Then, it loops through gameObjects and deletes and erases from gameObjects any Actors that have died. Lastly, the status text is updated. This is done using stringstream, where it lets int variables that need to be displayed (such as the infection status) become readable as strings, as well as handling the requirement of the score being displayed as 6 zeroes by default. This function is virtual because it is present in the GameWorld class. This function is in StudentWorld because it pertains to the game as a whole, as opposed to a specific actor.

**virtual** **void** cleanUp();

This function deletes Penelope and sets her to a nullptr if she is not already null. It also deletes all Actors from gameObjects by looping through gameObjects and deleting each Actor. This function is virtual because it is present in the GameWorld class. This function is in StudentWorld because it pertains to the game as a whole, as opposed to a specific actor.

**bool** checkDir(Direction dir, Actor\* act);

Checks to see if an Actor can move in Direction dir. It

works by looping through the list of Actors, and if an Actor can block other Actors, it checks its position to make sure their positions (with respect to SPRITE\_WIDTH and SPRITE\_HEIGHT) don’t overlap. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** exitOverlap(**int** x, **int** y);

Main code for exits. It loops through the list of Actors, and if it finds a citizen, increases the score, sets citizen to dead, and plays the correct sound. It then checks if all the citizens are dead; if so, it sets StudentWorld’s status as allowing Penelope to exit. This function is not virtual because there are no derived classes in StudentWorld, thus there are no classes that need to use the function. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** pitFlameProtocol(**int** x, **int** y, Actor \*act);

Since pits and flames have the same procedure for a human or zombie overlapping with them, I created one function that would be called by the Pit and Flame classes. First, it loops through gameObjects; if an Actor is a Citizen, it’ll call the function deadCitizenProtocol (which takes care of what needs to be done if a citizen is dead). If the Actor is a zombie, the function deadZombieProtocol (which takes care of what needs to be done if a zombie is dead) is called, and the function determines if the zombie will drop a VaccineGoodie. It does so by using the randInt function to generate an int that will determine whether or not it should drop a goodie, and where the goodie should go. For the direction picked, it checks to see if the goodie would overlap with any other actors by looping through gameObjects and comparing coordinates. If the goodie can be placed, a new VaccineGoodie object is created and added to gameObjects. This function is in StudentWorld because it needs access to gameObjects to function properly. If the Actor that overlaps with the pit/flame is a landmine, it calls the function landmineFlameProtocol, which handles what a landmine should do in the case that a flame overlaps with it. If a goodie overlaps, the goodie “dies”, so it will be removed from the game. Lastly, it checks if Penelope overlaps; if so, deadPenelopeProtocol (which takes care of what needs to be done if Penelope is dead) is called. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** infect(**int** x, **int** y);

This function sets the infection status of Penelope and Citizens to true, given if they overlap with the vomit coordinates sent to them. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** infectedCitizenProtocol(**int** x, **int** y);

Contains code to account for a citizen whose infection status has reached 500. It plays the “SOUND\_ZOMBIE\_BORN” sound, decreases the score, and uses randIt to determine whether or not a smart or dumb zombie should be born. It adds the correct type of zombie to gameObjects. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** moveCitizen(**int** x, **int** y, Citizen\* c);

This function handles all the code for how a citizen should move. First, it determines the distance between the citizen and Penelope, and determines the closest zombie and its distance between the citizen and the zombie using the closeZombie helper function. If Penelope’s distance is closer, the citizen determines which direction to run based on Penelope’s coordinates, and moving 2 pixels in a direction that would move the citizen closer to Penelope. If the citizen does not share an axis with Penelope, it will randomly pick a direction that would bring it closer to Penelope. If Penelope’s distance is larger than the closest zombie, and the closest zombie is within 80 pixels, the citizen determines which direction will lead it away from the closest zombie if it were to move in that direction. It then checks to see if it can move in a direction away from the zombie; if so, it will set the direction and move. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** landmineFlameProtocol(**int** x, **int** y, Direction dir);

This function handles how a landmine should act in the case that something overlaps with a landmine. First, SOUND\_LANDMINE\_EXPLODE is played, and a flame is added in the same spot as the landmine. It then checks if flames can be added around the landmine; if so, a new flame is added in that spot as well. Lastly, a Pit is added in the same spot the landmine is in. This function is in StudentWorld because it needs access to gameObjects to function properly.

**bool** getVacGoodie(**int** x, **int** y);

This function handles what should happen should Penelope pick up a VaccineGoodie. It calls the helper function goodieOverlap, which covers what should happen if Penelope picks up any kind of goodie, then increases the number of vaccines Penelope has. This function is in StudentWorld because it needs access to Penelope to function properly.

**bool** getGCGoodie(**int** x, **int** y);

This function handles what should happen should Penelope pick up a GasCanGoodie. It calls the helper function goodieOverlap, which covers what should happen if Penelope picks up any kind of goodie, then increases the number of flames Penelope has. This function is in StudentWorld because it needs access to Penelope to function properly.

**bool** getLMGoodie(**int** x, **int** y);

This function handles what should happen should Penelope pick up a LandmineGoodie. It calls the helper function goodieOverlap, which covers what should happen if Penelope picks up any kind of goodie, then increases the number of landmines Penelope has. This function is in StudentWorld because it needs access to Penelope to function properly.

**void** explode(**int** x, **int** y, Direction dir, Landmine\* act);

This function handles what happens to certain Actors when a Landmine explodes. If Penelope set off the landmine (determined by whether or not she overlaps with it using a helper overlap function), the landmine is set to dead, the function deadPenelopeProtocol is called, and landmineFlameProtocol is called. It then loops through gameObjects. If a citizen set off the landmine (determined by whether or not they overlap with it using a helper overlap function), the landmine is set to dead, the function deadCitizenProtocol is called, and landmineFlameProtocol is called. If a zombie set off the landmine (determined by whether or not they overlap with it using a helper overlap function), the landmine is set to dead, the function deadZombieProtocol is called, and landmineFlameProtocol is called. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** addALandmine(**int** x, **int** y);

This function adds a landmine to gameObjects using coordinates given to it. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** smartZDir(**int** x, **int** y, SmartZombie\* sz);

This determines the direction for smart zombies. It sets the movement plan distance if it’s 0, then determines the nearest human by looping through gameObjects and calculating the Euclidean distance, and calculating the same for Penelope. The program keeps track of the smallest distance in an int, as well as the Actor that had the smallest distance in a pointer to that Actor. If the Euclidean distance is greater than 80 pixels, the zombie picks a random direction to move in. If not, it determines which direction to move by identifying the coordinates of the closest human and comparing it with its own coordinates. If the human is in the same x or y axis, the zombie will set itself in the direction that will take it closer. If not, it picks a random axis to follow and sets the direction that will place itself closer to the human.

**void** penFlames();

This function covers what needs to happen when Penelope uses her flamethrower. First, it determines the position of the flames based on Penelope’s direction and coordinates. Then, it checks if there is an actor that can block flames; if so the function returns because a flame would not be able to be created. If a flame can be created, it adds the flame to gameObjects. This loops 3 times, because Penelope’s flamethrower produces 3 flames. This function is in StudentWorld because it needs access to gameObjects to function properly.

**void** goVomit(**int** x, **int** y, Direction dir, Zombie\* z);

This function handles what happens when a zombie vomits on a human. Depending on the zombie’s direction it computes what the coordinates of the vomit would be. It then goes through gameObjects; if there is a citizen that would overlap with the vomit, the zombie decides whether or not to vomit randomly using randInt. If the zombie decides to vomit, the vomit sound is played, and a new vomit object is added to gameObjects. It then checks Penelope; if she would overlap with the vomit and the zombie decides to vomit, the sound is played and the new vomit object is created. This function is in StudentWorld because it needs access to gameObjects to function properly.

**virtual** ~StudentWorld();

Calls the cleanUp function, as described above.

Actor class functions:

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

This function is virtual because each derived class has its own version of it.

For Penelope:

If Penelope is dead, the function returns. If she is infected, her infection status increases. If her infection status reaches 500, she is set to dead, StudentWorld is called to play the SOUND\_PLAYER\_DIE sound and to decrement the number of lives. It then checks if a player pressed a key. If they pressed space and Penelope has flames, her number of flames decreases, the fire sound is played, and the function penFlames is called. If tab is pressed, a landmine is created. If enter is pressed, her infection status is reset. If a directional key is pressed, and Penelope can move in that direction, she will move in that direction. This function is in the Penelope class because it was required according to the spec.

For Citizen:

If the citizen is dead, the function returns. If it is infected, its infection status increases. If its infection status reaches 500, it is set to dead and the function infectedCitizenProtocol is called. The citizen switches paralysis status, and the function moveCitizen is called. This function is in the Citizen class because it was required according to the spec.

For Wall:

Does nothing because walls do nothing. This function is in the Wall class because it was required according to the spec.

For Exit:

Calls the function exitOverlap. This function is in the Exit class because it was required according to the spec.

For Pit:

Calls the function pitFlameProtocol. This function is in the Pit class because it was required according to the spec.

For Flame:

If the flame is dead, it returns. If two ticks have passed, the flame is dead. The function calls pitFlameProtocol. This function is in the Flame class because it was required according to the spec.

For Vomit:

If the vomit is dead, it returns. If two ticks have passed, the vomit is dead. The function infect is called. This function is in the Vomit class because it was required according to the spec.

For Landmine:

If the landmine is dead, the function returns. It decrements the amount of safety ticks, and if there are no more safety ticks, the function is set to active and the function explode is called. This function is in the Landmine class because it was required according to the spec.

For VaccineGoodie:

If the goodie is dead, it returns. The function calls the function getVacGoodie, and if that returns true, the goodie is dead. This function is in the VaccineGoodie class because it was required according to the spec.

For GasCanGoodie:

If the goodie is dead, it returns. The function calls the function getGCGoodie, and if that returns true, the goodie is dead. This function is in the GasCanGoodie class because it was required according to the spec.

For LandmineGoodie:

If the goodie is dead, it returns. The function calls the function getLandmineGoodie, and if that returns true, the goodie is dead. This function is in the LandmineGoodie class because it was required according to the spec.

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

Returns the StudentWorld associated with the actor. This is in Actor because Actors need to be able to access the world they are associated with.

**bool** getIfDead() **const** {**return** isDead;}

Returns whether or not an Actor is alive. This is in Actor because Actors need to be able to access their own alive/dead status.

**void** setDead() {isDead = **true**;}

Sets actors to dead. This is in Actor because Actors need to be able to change their own alive/dead status.

**virtual** **bool** canBeInfected() **const** {**return** **false**;}

Returns whether or not Actors can be infected. This function is virtual because each Actor has its own characteristics; some can be infected, and some cannot. This is in Actor because it is associated purely with the Actor class it is in.

**virtual** **bool** canBeDamaged() **const** {**return** **true**;}

Returns whether or not Actors can be damaged. This function is virtual because each Actor has its own characteristics; some can be damaged, and some cannot. This is in Actor because it is associated purely with the Actor class it is in.

**virtual** **bool** canBlock() **const** {**return** **false**;}

Returns whether or not Actors can block other objects. This function is virtual because each Actor has its own characteristics; some can block, and some cannot. This is in Actor because it is associated purely with the Actor class it is in.

**virtual** **bool** canBlockFlames() **const** {**return** **false**;}

Returns whether or not Actors can block flames. This function is virtual because each Actor has its own characteristics; some can block flames, and some cannot. This is in Actor because it is associated purely with the Actor class it is in.

**virtual** **bool** canBePickedUp() **const** {**return** **false**;}

Returns whether or not Actors can be picked up. This function is virtual because each Actor has its own characteristics; some can be picked up, and some cannot. This is in Actor because it is associated purely with the Actor class it is in.

**virtual** **void** setInfect(**bool** b) {}

Sets infection status to true or false. Doesn’t have to be virtual but is virtual for good practice. This is in actor because Actors need to be able to change their infection status.

**virtual** **void** deathPoints() {};

Awards the points upon the death of the zombie. This is virtual because zombies give different points when they die based on what type of zombie it is. This is in actor because it is associated with the actor class it is a part of.

**virtual** **bool** getIfParalyzed() {**return** isParalyzed;}

Returns if a Human is paralyzed. Is virtual for good practice. In Agent because Zombies and Citizens use it.

**virtual** **void** switchParalysis() {isParalyzed = !isParalyzed;}

Changes paralysis status from false to true and vice versa. Is virtual for good practice. In Agent because Zombies and Citizens use it.

**virtual** **int** getMPD() {**return** movementPlanDistance;}

Returns the movement plan distance. Is virtual for good practice. In Zombie because both dumb and smart zombies use it.

**virtual** **void** setMPD (**int** mpd) {movementPlanDistance = mpd;}

Sets the movement plan distance. Is virtual for good practice. In Zombie because both dumb and smart zombies use it.

**virtual** **bool** getIfInf() **const** {**return** isInfected;}

Returns whether or not a human has been infected. Is virtual for good practice. Is in Human because Penelope and Citizens use it.

**virtual** **void** setInfectStat(**int** num) {infectionStatus = num;}

Sets infection status. Is virtual for good practice. In Human class because Penelope and Citizens use it.

**virtual** **int** getInfStatus() {**return** infectionStatus;}

Returns infection status. Is virtual for good practice. In Human class because Penelope and Citizens use it.

**void** incVaccines() {numVaccines++;}

Increments the number of vaccines. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**void** incFlames() {numFlames += 5;}

Increments the number of flames. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**void** incLandmines() {numLandmines += 2;}

Increments the number of landmines. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**int** getVaccines() **const** {**return** numVaccines;}

Returns the number of vaccines. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**int** getFlames() **const** {**return** numFlames;}

Returns the number of flames. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**int** getLandmines() **const** {**return** numLandmines;}

Returns the number of landmines. Is not virtual because it is only present in the Penelope class. In Penelope because only Penelope needs it.

**virtual** **void** setActive() {isActive = **true**;}

Sets landmine status to active. Is virtual for good practice. In landmine class because landmines have an active status.

2.

Functionality I failed to finish is the movement of the citizens. When the citizens start following Penelope, they end up moving onto Penelope. While testing my program I found out that certain bools that I wrote in Actor class didn’t get read properly by StudentWorld in the derived classes, so I had to rewrite some of the bool statements in the derived classes.

3.

For inheritance rules I followed the suggested design from the spec. I used a list for my actors because it was easier to use. I put functions in StudentWorld when I had to use my list of actors. As far as assumptions: I wasn’t sure if zombies can move over the exit, so I let them cross over the exit. The zombies don’t do anything when they’re there though. Also, when the citizens are running away from zombies, if the citizen is blocked from moving in the opposite direction that the zombie is walking towards, the citizen will check the remaining two directions – so if the zombie is moving to the left towards the citizen and the citizen is blocked from moving to the right, the citizen checks if it should move up or down, and it does so by checking if the zombie is below or above it. If the zombie is in the same y coordinate, I let the citizen default to trying to move downward. Conversely, the citizen also defaults to moving leftward.

4.

I tested my StudentWorld class by using multiple cout statements that keep track of where the code is. While I played the game I would do multiple things that would cause certain code to be run (I killed citizens to make sure the status bar showed a negative score correctly).

I tested my Actor class by commenting out the addition of objects into the level in StudentWorld’s init to keep track of the specific actor I wanted to look at. So when I tested smart zombie, I took out the code that added dumb zombie so I could easily see when I played the game if I implemented smart zombie correctly. I also used cout statements that keep track of certain variables (for example, if Penelope grabbed a gas can, I printed “got gascan” and the number of flames she had when she used it.

Specific testing for each Actor class:

DumbZombie:

I tested dumb zombies by playing the game multiple times, and running Penelope and Citizens close to it to make sure they vomit properly and that they didn’t follow Penelope and Citizens. I watched gameplay to make sure they moved around randomly and didn’t run into walls or use exits.

SmartZombie:

I tested smart zombies by setting up the level so that none of the dumb zombies would be there. I ran Penelope close to the zombie to make sure it followed her and vomited on her. I watched gameplay to make sure they moved around randomly if far from citizens and Penelope and didn’t run into walls or use exits.

Penelope:

I tested Penelope by moving her close to certain objects to make sure she didn’t overlap with them. I checked direction keys to make sure she moved properly. I tested her ability to fire flames, drop landmines, and use vaccines by collecting the goodies and pressing the proper keys. I tested her infection status by getting close to a zombie so it would infect her.

Citizen:

I tested citizen by moving Penelope towards them to make sure they followed her, and waiting for zombies to move towards them to test whether or not they were infected properly, and that they ran away from zombies properly.

Wall:

I shot flames at walls to make sure it blocked them and ran Penelope into walls to make sure she couldn’t go past them.

Exit:

I shot flames at them to make sure they didn’t destroy them, I saved citizens to make sure they left properly, I finished levels to make sure Penelope left properly, and I waited for zombies to be around exits to make sure they didn’t vomit on them.

Pit:

I led Penelope, Citizens, and Zombies to Pits to make sure they fell through them. I set of landmines to make sure pits were created after they died.

Flame:

I made Penelope shoot a wall to make sure it didn’t overlap when it wasn’t supposed to. I fired flames at citizens, zombies, and vaccines to make sure they were destroyed properly. I also set off landmines to make sure those flames were created properly, and disappeared after 2 ticks.

Vomit:

I ran Penelope and citizens towards Zombies to make sure Zombies properly vomited on them, the vomit was in the right spot, the vomit died after two ticks, and that Penelope and Citizens were infected. I also watched gameplay to make sure vomit wouldn’t show up at improper times.

Landmine:

I made Citizens, Penelope, and Zombies overlap with it so it worked properly for all of them. I made Penelope overlap with it before and after it was supposed to be active to make sure its active status was correct; I shot flames at it to make sure it exploded properly.

VaccineGoodie:

I made Penelope use her flamethrower on it so it would be destroyed. I made Penelope collect it and use it so those functions and the number of points would happen properly.

GasCanGoodie:

I made Penelope use her flamethrower on it so it would be destroyed. I made Penelope collect it and use it so those functions and the number of points would happen properly.

LandmineGoodie:

I made Penelope use her flamethrower on it so it would be destroyed. I made Penelope collect it and use it so those functions and the number of points would happen properly.