1. Description of Public Functions

\*\* All destructors are not listed and designated virtual since inheritance and polymorphism is utilized. They all do not do anything even though they have a dynamic variable, which points to a StudentWorld object. This is the case since I did not want to delete the world once a single object dies. \*\*

**class** StudentWorld : **public** GameWorld

Actor(StudentWorld\* world, **int** imageID, **int** startX, **int** startY, Direction d, **double** size, **int** depth);

Actor(StudentWorld\* world, **int** imageID, **int** startX, **int** startY);

I added 2 constructors, one for where the size and depth should be specified, and the default case, which has a size of 1. Objects are set to not visible by default

TunnelMan\* getTM();

This function returned a pointer to the tunnel man so other classes can access its coordinates or decrement its hp. I decided to do this instead of perform these operations inside the StudentWorld class since it made the most sense. More specifically, for the yell function of the protester, it made the most sense to decrement the hp of the tunnel man in that class rather than in the StudentWorld class, at least to me.

**bool** isDiggable(**int** x, **int** y, Actor::Direction d, **bool** isTM);

This function returned a bool that determined whether or not the object was next to earth. If the object was the tunnelman, it would be able to dig through. This was also used to detect if there was earth below a boulder.

**bool** nextToBoulder(**int** x, **int** y, Actor::Direction d);

This function returned a bool that determined whether or not the object was next to a boulder.

**bool** collidedwithTM(**int** x, **int** y, **double** distance);

This function was used to determine collision of a TM with a goodie or a boulder.

Protester\* collidedwithProtester(**int** x, **int** y);

This function was used to determine the collision of a protester with a gold nugget or boulder.

**double** getDistance(**int** x1, **int** y1, **int** x2, **int** y2);

This function was used to determine the distance between two objects.

**void** decBarrels();

This function was used to decrease the number of barrels of oil left as the number of barrels is a private variable of StudentWorld.

**void** decProtesters();

This was used to decrement the number of protesters if an event related to this took place. I had a numProtesters private variable to keep track of the max number of protesters for a level.

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

This was used to add special actors such as squirts inside the player class

**void** makeDistributablesVisible(**double** distance);

This was used by the sonar class to make barrels and nuggets (which I put into the distributable class) visible

**void** moveTowardsExit(Protester\* p);

BFS method for the protester to return to the exit. I implemented the BFS search with a queue and tracked the path to the exit by using a map that associated a coordinate with the coordinate taken before it.

**void** sensePlayer(Protester\* p);

BFS method for the hardcore protester to track down the tunnel man. The implementation was similar to that of the moveTowardsExit() function, but I used the coordinates of the tunnel man instead of the exit

Actor::Direction findDirectiontoTM(**int** x, **int** y);

This was used to see if the protester had direct line of sight to the tunnel man.

**bool** validProtesterMove(**int** x, **int** y, Actor::Direction d);

This was used to determine if the protester could move in a specific direction, unobstructed by boulders or earth.

**class** Actor : **public** GraphObject

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

This should be a virtual function since each class derived from actor has a different doSomething. Since there is not “default” do something, this was made pure virtual.

**bool** isAnnoyed();

Getter function of the int m\_annoyed variable. This is a variable universal to all classes derived from Actor, which is why it belongs here.

**void** setAnnoyed();

Setter function of the int m\_annoyed variable.

StudentWorld\* getWorld();

Getter function for StudentWorld\* m\_world pointer variable. This is a variable universal to all classes derived from Actor, which is why it belongs here.

**int** getTicks();

Getter function for the int m\_ticks variable. Most classes have special functionality that depends on the number of ticks elapsed. Furthermore, these classes are only related in the fact that they are Actors (ex: Protesters and Boulders have tick functionality), so I decided to implement a tick private variable in Actor.

**void** incTicks();

Setter function for the int m\_ticks variable.

**virtual** **bool** isDistributable();

Virtual function - default returns false since all objects are not distributables unless otherwise designated. If the object is a distributable, override and return true. Used to properly distribute distributables in the StudentWorld init function.

**class** Earth : **public** Actor

Earth(StudentWorld\* world, **int** x, **int** y);

Call’s actor’s constructor with size .25 and depth 3.

**virtual** **void** doSomething();

Earth has an implementation (just {} ) to prevent it from becoming an ABC.

**class** Sentient : **public** Actor

BACKGROUND: Sentients are what the tunnel man and protesters are derived from. As the name implies, they are sentient beings and have an hp variable to reflect that.

Sentient(StudentWorld\* world, **int** imageID, **int** x, **int** y, Direction d, **int** hp);

Since sentient is the base class of many different classes, an imageID is required in its constructor. Furthermore, since the tunnelman and protester initially face different directions, a direction parameter is also needed. This is also the case with HP

**int** getHP();

Getter function for the int m\_hp variable

**void** decHP(**int** i);

Setter function for the int m\_hp variable.

**virtual** **void** playDigSound(Direction d);

By default this does nothing, which is why I made it virtual. Only the tunnel man is allowed to play the dig sound.

**void** directionalMovement(Direction d);

This is where all the sentient movement is handled, including the dig sound playing. If its a protester moving, it does not play a sound whereas the tunnel man can play a sound. Since this is universal to both the tunnel man and protesters, it is not designated as virtual.

**class** TunnelMan : **public** Sentient

TunnelMan(StudentWorld\* world);

Initializes the items in the tunnel man’s inventory

**virtual** **void** playDigSound(Direction d);

This is designated as virtual to keep in line with good OOP practices. This overrides the default Sentient playDigSound and allows the tunnel man to play a dig sound.

**virtual** **void** doSomething();

Designates what the tunnel man is to do during each tick. Includes taking input from the player and deciding the appropriate course of action. Designated virtual for good OOP practices.

**int** getGold();

Getter function for the int m\_numGold variable, which keeps track of how much gold is in the player’s inventory. Used for the display text.

**int** getSonar();

Getter function for the variable that keeps track of how much sonar the player has. Used for display text.

**int** getSquirts();

Getter function for the variable that keeps track of how much ammo the tunnel man has. Used for display text.

**void** addToInventory(**int** ID);

Used to add to the tunnel man’s inventory once a collision is detected. Differs based on the ID of the object.

**class** Squirt : **public** Actor

Squirt(StudentWorld\* world, **int** x, **int** y, Direction d);

Sets m\_coord, which is the max distance a squirt can travel, to 4.

**virtual** **void** doSomething();

Designates what the squirt is to do during each tick. Able to move forward and damage protesters. Made virtual for good OOP practices.

**class** Protester : **public** Sentient

BACKGROUND: Protesters are divided into hardcore and regular, giving birth to the general protester class.

Protester(StudentWorld\* world, **int** imageID, **int** hp);

Initializes all special protester variables such as ticks between moves, ticks since yell, ticks since a perpendicular turn was made, and number of squares to move in current direction. There are no getter or setter functions for these variables since no other class should alter these values.

**virtual** **void** doSomething();

Designates what the protester is to do during each tick. This includes but is not limited to various functionalities such as moving towards the player if in line of sight, random movement, and moving back to the exit. Made virtual for good OOP practices.

**virtual** **bool** trackPlayer();

Only hardcore protesters are able to track the player. Therefore, this is designated as virtual since the default case is that the regular protester cannot track the player. The importance of this is for the doSomething() function, which differs between the regular and hardcore protesters only in this function. This allows for doSomething() to remain the same between both types of protesters to reduce duplication of code.

**void** intersectionDirection();

Sets the direction to change to once the protester encounters an intersection.

**void** resetNumSquares();

Resets the number of squares to move in the current direction using rand().

Direction generateDirection();

Randomly generate a direction using rand().

**void** getSquirted(**int** i);

Differs from decHP in the fact that protesters get stunned when they are shot by the tunnel man. Could have overridden decHP but decided to do it this way for clarity.

**void** stun(**int** ticks);

Stun the protester if they are shot.

**virtual** **void** bribe();

Bribe the protester if they encounter a player dropped gold nugget. This is designated as virtual since the regular and hardcore protesters have differing responses to encountering a player dropped gold nugget

**bool** decideToMove();

This takes into account the resting and non resting ticks of the protester and decides when it is appropriate for the protester to move

**bool** facingTM();

If the protester is facing the tunnel man, it is allowed to yell. This function calculates when the protester is facing the tunnel man.

**void** yell();

Annoy the tunnel man by yelling if close enough

**bool** getState();

Returns whether or not the protester is in a “leave the field” state.

**void** setLeave();

Setter function for the leave the field variable

**class** RegularProtester : **public** Protester

Protesters have no special functionality except for the fact that they have 5 hp compared to hardcore protesters. I made a special regular protester class to emphasize the difference between a hardcore protester, but I realize that this is not needed.

RegularProtester(StudentWorld\* world);

Calls protester’s constructor with a protester image and hp.

**class** HardcoreProtester : **public** Protester

HardcoreProtester(StudentWorld\* world);

Calls protester’s constructor with a hardcore protester image and hp.

**virtual** **void** bribe();

Hardcore protesters, when bribed, stare at the nugget, which is why the original bribe() function is overriden. This is set as virtual to keep in line with good OOP practices.

**virtual** **bool** trackPlayer();

Designated as virtual to keep in line with good OOP practices. Allows the hardcore protester to track the player if near enough.

**class** Distributable : **public** Actor

BACKGROUND: Distributables are objects that are distributed throughout the earth. Used moreso to separate other objects from sentients

Distributable(StudentWorld\* world, **int** imageID, **int** x, **int** y, Direction d, **double** depth);

Calls actor’s constructor with the specified parameters and a size of 1.

**virtual** **bool** isDistributable();

Getter function for the isdistributable variable, which is set to true for distributable objects. Set as virtual for good OOP practices.

**class** Boulder : **public** Distributable

Boulder(StudentWorld\* world, **int** x, **int** y);

Calls Distributable’s constructor with the specified parameters. Initializes the m\_waiting variable, which determines whether or not it is time for the boulder to fall and the m\_playSound variable, which allows the boulder to play its falling sound ONCE when its falling

**virtual** **void** doSomething();

Special function for what the boulder is to do during each tick, such as what to do when the earth below it is dug up and collision incidents between the protester/tunnel man. Set as virtual to keep in line with good OOP practices.

**class** Barrel : **public** Distributable

Barrel(StudentWorld\* world, **int** x, **int** y);

Calls Distributable’s constructor with the specified parameters

**virtual** **void** doSomething();

Special function for what the barrel is to do during each tick. Set as virtual to keep in line with good OOP practices.

**class** Gold : **public** Distributable

Gold(StudentWorld\* world, **int** x, **int** y, **bool** state);

Calls Distributable’s constructor with the specified parameters. Takes in a bool for its state to determine if it can be picked up by a protester or tunnel man.

**virtual** **void** doSomething();

Special function for what the gold is to do during each tick. This includes what happens if it is picked up by the tunnel man or protester based on its state. Set as virtual to keep in line with good OOP practices.

**class** Spawnable : **public** Distributable

BACKGROUND: Spawnable items are items that spawn randomly throughout the earth.

Spawnable(StudentWorld\* world, **int** imageID, **int** x, **int** y);

Initializes the m\_maxTicks variable, which determines how long the object should exist

**void** spawnableDoSomething(**int** score);

This should be an ABC, so it does not override Actor’s doSomething(). Rather it has its own special spawnable doSomething() function that gives the similar functionality to its derived classes which differ only in the amount they increase the score.

**class** Sonar : **public** Spawnable

Sonar(StudentWorld\* world);

Calls Spawnable’s constructor. X and Y are constant at 0 and 60 respectively

**virtual** **void** doSomething();

Virtual to keep in line with good OOP practices. Calls spawnableDoSomething() with its unique score value.

**class** Water : **public** Spawnable

Water(StudentWorld\* world, **int** x, **int** y);

Calls Spawnable’s constructor with the specified parameters as a water pool can spawn at varying locations in the world.

**virtual** **void** doSomething();

Virtual to keep in line with good OOP practices. Calls spawnableDoSomething() with its unique score value.

2. Known Bugs

I implemented all of the functionality of the spec to the best of my ability. However, a bug that I tried debugging but I can’t figure out is that the hardcore protesters sometimes act odd when the tunnel man is right next to a boulder. More specifically, they don’t seem to track the tunnelman when it is right next to a boulder. Otherwise, when the tunnelman is digging through the earth, hardcore protesters are able to properly track the tunnelman and leave using the BFS maze searching method. Another bug is that the tunnelman does not pause after turning directions similar to the sample game, which I’m also not sure how to fix.

3. Design Assumptions

One case of ambiguity was number of protesters to add since the level is multiplied by 1.5, so I rounded down for odd levels. Another case of ambiguity was how many sonars to add, which isn’t listed in the spec but was 2 inside the sample game.

Additionally, there is what the ticks since perpendicular turn variable should be. I was confused on whether it should start at 0 or a large value to allow the protester to immediately make a perpendicular turn. I decided to set it to a large placeholder value

4. Testing

**Note: When I reference refreshing, I mean pressing escape and restarting without decrementing lives so I could constantly observe how the classes function.**

**class** StudentWorld : **public** GameWorld

Most of the testing of the student world functions was done by observing the interaction of other objects in the game with the tunnel man. For example, one of the functions was determining if the tunnel man was next to a boulder, and this was elucidated in the boulder testing.

**class** Actor : **public** GraphObject

I tested the general actor class functions and variables by seeing if objects were removed after becoming annoyed. Tick functionality was tested by freezing and counting tick by tick for relevant classes.

**class** Earth : **public** Actor

I tested this class by digging around and seeing if the earth objects were removed when the tunnel man dug around. I also verified that the earth was properly sized at .25 when digging around.

**class** Sentient : **public** Actor

Sentient functionality was tested by making sure that only the tunnel man could dig through earth and play dig sounds. HP getter and setter functionality was tested by observing the health % in the display text as well as keeping a mental count of protester and hardcore protester HP.

**class** TunnelMan : **public** Sentient

Testing for the tunnel man class lied in making sure that all inputs worked. For one, the tunnel man was able to move throughout the whole earth without moving past the edges or through boulders. For consumable items, I tested the functions by making the inventory infinite and spamming sonars, the water gun, and gold drops. I also ensured that the display text was properly being updated after each consumable was used.

**class** Squirt : **public** Actor

I tested the squirt class by making the number infinite in the tunnel man’s inventory. I tested it by digging a large 20 x 20 area in the earth and shooting all directions to verify that the length is the same for all directions. I also tested what it looked like when shooting boulders and the earth, at 4, 3, 2, and 1 spaces away to ensure proper functionality. I also shot protesters to verify that they were properly decrementing the health of the protesters.

**class** RegularProtester : **public** Protester

For the protester, testing was performed by removing the ability of the protester to move on its own and only move when the tunnel man was in the line of sight. This allowed for me to comprehensively test each parameter, also assisted by freezing and observing each tick. For example, I could lure the protester to under a boulder and observe whether or not it became annoyed and ran back. I further tested its ability to run back by luring it to a random corner through a convoluted maze and killing it. To test directionality, I decreased the tick per perpendicular turn to 5, removed the ability of the protester to move, and lured the protester into an open space. From there, I observed if it was properly changing directions (if it was spinning in all directions). In order to test the ability of the protester to follow the tunnel man when it is in its line of sight specifically, I brought the protester deep into the Earth and made sure it could not “see” the tunnel man if there was only 1 Earth object blocking it.

Once I added the ability for the protester to move independently, I set the number of spaces to move in the current direction to 1000 so it would just walk straight. Once it hit once of the corners, I observed whether or not it properly changed directions. I then set this to its normal random value between 8 and 60 and observed whether it was moving similar to the sample game.

**class** HardcoreProtester : **public** Protester

The hardcore protester had similar functionality to a regular protester, but differed in the fact that it could track the player. I tested this by changing the chance of hardcore protesters to spawn to 100% and allowing it to track the tunnel man from any distance in the game. I set a slight delay of spawning protesters so I could move to a proper position and dug up a complex maze and observed if the hardcore protester could make its way towards the tunnel man. Once killing the hardcore protester, I also observed if it was moving back to the exit properly.

**class** Distributable : **public** Actor

I made sure that distributables were spawning 6 blocks from each other by making the barrels, nuggets, and boulders initially visible. I removed all other objects to isolate these. I then altered the cleanUp() function to not dispose of these objects and constantly refreshed and observed as the number of distributables increased. This sounds odd, but I knew that it worked properly once my game crashed. More specifically, I used a while loop to constantly generate new x and y values for distributables, and if the loop ran infinitely, it indicated that new x and y values were infinitely being generated, all invalid due to the lack of space in the world.

**class** Boulder : **public** Distributable

Boulders were tested by first removing all other objects so they could be independently tested. I checked that they were all spawning above 20 by spawning over 50 boulders each instance and refreshing the world many times. I tested the borders of the boulder using the tunnel man and the falling functionality by freezing and iterating tick by tick. It was difficult to time properly, but if it fell in a range of around 25-30, I thought it was fine. Furthermore, in order to test if it would die when it hit a boulder, I manually spawned boulders on top of each other and broke the earth underneath. To see if it was properly disposed of when it hit the edge of the screen, I dug up all of the earth underneath the boulder.

**class** Barrel : **public** Distributable

In order to test barrels, I set them to visible. I also set that when I was near the barrels, they would turn invisible so I could properly fine tune the distances. This is essentially the same functionality as having them first invisible and seeing if they turn visible, except I know where each and every barrel is. I also ensured that the number of barrels was properly being decremented in the display text once a barrel was picked up.

**class** Gold : **public** Distributable

Detecting when gold became visible was similar to the testing of the barrel. To test the dropping functionality, I lured a protester to a gold nugget that only a protester could pick up. I then dropped a gold nugget in front of a protester, ensuring that the tunnel man could not pick it up and only the protester could. Additionally, I also verified that the protester could not pick up the gold nugget when it was not already picked up by the tunnelman

**class** Spawnable : **public** Distributable

**class** Sonar : **public** Spawnable

**class** Water : **public** Spawnable

Testing for these classes were combined due to similar functionality. I tested that sonar objects are being spawned properly by adjusting the rate of spawnables spawning. I created a sonar and water pool count and incremented those when a respective item was spawned in. I printed the ratio between the counts in my cleanUp() function so the ratio could be displayed after refreshing the level and not constantly each tick. Once each of these items were picked up, I ensured that the display text was properly updating.