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 blah() function in my base class because all Actors in TunnelMan must have a blah function, and each type of actor defines their own special version of it.”

**StudentWorld class:**

//Functions required to be implemented

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

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

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

The above 3 functions are specified in the project specifications in terms of their functionality. The init function first creates a TunnelMan object whose pointer is stored in this class as well as a 2-D Earth array populated with Earth except for the tunnel. It then calls upon initBoulder, initBarrel, and initGold helper functions to distribute the boulder, oil barrels, and gold nuggets. The move function’s pseudocode is provided in depth in the project specification page 19. cleanUp essentially deletes all the dynamically allocated objects which were stored in a vector of Actor pointers. It does this using a loop which deletes the object the pointer points to and then deletes the pointer in the vector itself. Vectors were used due to their dynamic sizing properties.

//Functions used within init() to initialize

**void** initBoulder();

**void** initBarrel();

**void** initGold();

The above 3 functions all perform the same task but for different objects. Each of the boulders, barrels, and gold nuggets are spawned in certain quantities based on the level and each have restrictions as to where they can be spawned. They must also be spawned 6 unit radius away from every other object. These three functions accomplish the described task for the named objects specifically, ensuring that exactly the amount of boulders, barrels, and gold needed is spawned and not more.

//Functions used within move()

**void** removeDeadGameObjects(std::vector<Actor\*>& v);

**void** addGoodies();

**void** addProtesters();

The first function was created first in the Project warmup where I implemented removeOdd function using vectors and removeBad. This essentially removes any dead objects from the Actors vector. The addGoodies and addProtesters is included since these can spawn during gametime and thus must be include in move. They are spawned using the probability given in the spec sheet and are kept under the maximum number possible.

//Additional functions used by Actor objects

**bool** digEarth();

digEarth is a function called upon when the user makes TunnelMan move. It is used only by the TunnelMan and the function essentially removes the quantity of Earth which TunnelMan overlaps with. This function returns a Boolean value where true indicates digging was done and false indicates no digging was done by the TunnelMan.

**bool** checkBelowBoulder(**int**, **int**);

This function is used by the Boulder class but included in StudentWorld since the 2D Earth array must be scanned. It essentially checks whether there is Earth or not directly below the boulder, if there is Earth it returns true otherwise it returns false. This is useful in the Boulder class when it is in a falling state.

**bool** collisionTunnelMan(Actor\* boulder);

This function is used to check if a passed in boulder will collide with a TunnelMan (i.e. get near a radius of 3 when falling) and it does so using the radial distance function described below. Returns true if collision occurs, false if it will not occur.

**void** collisionProtester(**int** curX, **int** curY);

This function is used to check if a boulder will collide with a Protester (i.e. get near a radius of 3 when falling) and it does so using the radial distance function described below. Returns true if collision occurs, false if it will not occur. Note that we must pass in the coordinates of the boulder rather than a pointer (hence we must use getX(), getY()).

**void** sonarUsed(**int** curX, **int** curY);

This is called by the TunnelMan class when the user inputs a ‘Z’ or ‘z’. When this is called, the objects within a 12 unit radius are set to visible by the setVisible() function.

**void** shootProtester(**int** curX, **int** curY);

//Queue based maze solving algorithm

**void** createExitPath(**int** maze[61][61], **int** ex, **int** ey);

A maze array is passed in alongside the value of the endpoints and this function fills the maze array with numbers indicating how many steps away it is from the endpoint. This strategy of using a 2-D maze of integers to store steps and using a queue for breadth first search was suggested by Nima on GroupMe. The code’s implementation was inspired by the lecture slides on solving a maze using a BFS approach.

**void** createPathToTunnelMan(**int** maze[61][61], **int** tunnelX, **int** tunnelY);

This function essentially calls createExitPath but instead the coordinates of the tunnelMan are provided and a different maze is inputted.

//Used to calculate euclidean distance between two given coordinates

**int** radialDistance(**int** firstX, **int** firstY, **int** secondX, **int** secondY);

Given two coordinates, the difference between their x and y values are individually computed and then squared. These values are then added and the square root of these values is returned.

//We have two versions: one version checks for a SPECIFIED objectType, the other scans for any and all objects

**bool** checkForObjects(Actor \*p, **int** radius); //by default the radius will be 6.0

If an Actor pointer is passed in alongside a radius value (which is 6.0 by default) the function scans a radius to see if any other objects are present relative to the coordinates of the Actor pointer. This is used in many instances by the Actor classes.

**bool** checkForObjects(ObjectType type, Actor \*p, **int** radius);

An enum named ObjectType was created which is passed in for every Actor constructed and the enum consists of the names of each of the classes. This parameter overloaded function does the same thing as the function above it but it scans for a certain type of Actor which is passed into it. The function uses “getObjectType()” which is a function in the Actor class that tells you the type of an object and returns an enum ObjectType value.

**bool** checkForObjects(**int** curX, **int** curY, **int** radius); //by default the radius will be 6.0

The third and final overloaded checkForObjects takes in coordinates (in case we have not created an object yet) and scans in a provided radius to see if there are other objects. This function is used in init() since we must check for objects near a random coordinate placement point before we actually create our Actor object.

**bool** checkForEarth(**int** curX, **int** curY);

This function checks if the 4x4 grid surrounding the given curX and curY value (which are assumed to be lower left points of this grid) is free of Earth objects.

**bool** checkForBoulder(**int** curX, **int** curY);

This function checks specifically if there are no boulders in a 3 mile radius of the given coordinates.

**bool** pathObstructed(**int** curX, **int** curY);

The path obstructed function checks if the passed in values are within the grid of the game (i.e. it ensures the values are within 0 and 63) and then it calls checkForEarth and checkForBoulder and returns true if both values are true and false if either or both are false.

//Accessor/mutator functions

**int** getTicks();

Used to obtain the number of ticks passed by since init was called.

TunnelMan\* getTunnelMan();

Used to obtain a pointer to the TunnelMan object

**void** decBarrels();

Used to reduce the private member number of Barrels by 1

**void** decBoulders();

Used to reduce the private member number of Boulders by 1

**void** decProtesters();

Used to reduce the private member number of Protesters by 1

**void** insertObject(Actor\* O);

Used to insert an Actor pointer into the vector of Actor objects so that it appears in the game

**void** setProtesterTick();

Used to set the ticks between every new protester which is determined by a formula given on page 20 of the spec sheet.

**Actor class:**

//Constructor for Actor (for objects which have size and depth of 1.0 and 0 and must start facing rightward)

Actor(**int** imageID, **int** startX, **int** startY, StudentWorld\* world, ObjectType type, **int** health = 0 );

Actor(**int** imageID, **int** startX, **int** startY, Direction dir, **double** size, **unsigned** **int** depth, StudentWorld\* world, ObjectType type, **int** health = 0);

The above are constructors for the actor which is itself an abstract base class. When a derived class is thus created, this constructor is called and it forces the user to input the imageID, starting coordinates, StudentWorld pointer, and ObjectType which is an enum I created to tell apart the objects in the large vector in StudentWorld which contains all of these objects. The constructor also initializes all of the private member variables which are described later. Since Actor itself is derived from GraphObject, it passes in the size, depth (if required), imageID, and starting coordinates into GraphObject. The StudentWorld pointer is added as a private member which derived classes can access so that they can access the public functions of the StudentWorld.

//Destructor for Actor (virtual due to polymorphism)

**virtual** ~Actor();

This destructor has an empty body since no dynamic memory exists in this class.

//Status true means alive, else dead

**bool** getStatus();

An accessor to a private member called status which indicates whether something is alive or dead.

//Sets status to dead (i.e. false)

**void** setDead();

A settor which sets the status to false (i.e. dead)

//Pure virtual method used by almost every child

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

I decided to make this a pure virtual function since Actor itself is never required to be instantiated. This also forces every derived class to implement a doSomething function which the move() function in StudentWorld calls.

**ObjectType** getObjectType();

An accessor which returns the objectType private member.

//Accessor function to access + use StudentWorld passed into Actor

StudentWorld\* getWorld();

//Used by only the Protesters and TunnelMan

**int** getHitPoints();

Used to access a private member called hitPoints in actor which is initialized by passing in a health value when Actor is constructed. Note for non-living objects this is zero by default and never used.

**void** decHitPoints(**int** annoyance);

Used to deal damage to TunnelMan and Protesters. The function is included is in the actor class so that it can be called upon the Actor pointers that are stored in the vector STL container in StudentWorld.

**void** setHitPoints(**int** set);

A settor function to set the value of hitpoints to a specific number.

**bool** getLeave();

This is only relevant to protesters but it is added here since it needs to be accessed by Squirt and other objects when trying to deal damage to the protesters that exist in the vector STL container.

**TunnelMan class: derived from Actor**

//Constructor:

TunnelMan(StudentWorld\* myWorld);

Initializes a TunnelMan object with the starting amount of gold, squirts, sonar, health, etc. Since this is derived from Actor, a member initializer is used to initialize the Actor base class.

//Destructor

**virtual** ~TunnelMan();

Does nothing

//Inherited function from Actor (otherwise TunnelMan becomes ABC)

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

The doSomething() function uses user input as described on page 31 to move the TunnelMan and to make it use the SonarKit, the squirt gun, or drop a gold nugget.

For each direction (up, down, left, right) the TunnelMan checks if there is a boulder using the checkForBoulder function by first accessing getWorld(). If a boulder exists then it returns immediately, otherwise it moves one step in the inputted direction. Before the switch statements in the code is executed, the dig function is called to remove any earth that overlaps the Tunnelman. Once again, this dig function is accessed via getWorld().

When SPACE is pressed, if there is squirts in the squirt gun then a Squirt object is created which is placed 4 units away in the direction that the TunnelMan faces IF there is no Boulder or Earth in the way (this is checked using getWorld()->pathObstructed function). The sound of the water is also played and a squirt is decremented.

When TAB is pressed, if getGoldNuggets() is non zero we decrement it and drop a gold nugget initialized as temporary (by setting it’s dropped member value to true). We then decrement the amount of gold nuggets held by the TunnelMan object by 1.

When Z or z is pressed, the useSonar function is called from getWorld and the number of sonar kits available are decreased by 1.

//Accessor and Mutator functions

**int** getSquirts();

Returns the number of squirts available in squirt gun

**int** getSonar();

Returns the number of sonars available

**int** getGoldNuggets();

Returns the number of gold nuggets available

//The following is done when a goodie is picked up

**void** incSquirts();

**void** incSonar();

**void** incGoldNuggets();

Each of the above function increments the goody private members in the TunnelMan class, incSquirt, however, increases squirts by 5 rather than 1 since picking up a WaterPool increases squirts by 5.

**Protester class: derived from Actor**

//By default, health will be 5 and image will be Regular, we must specify when constructing hardcore to change these

Protester(StudentWorld\* myWorld, **bool** hardcore = **false**, **int** health = 5, **int** imageID = TID\_PROTESTER);

By default (i.e. when the protester is regular) it has a hardcore value of false, a health of 5, and an image ID of a regular protester. During construction, all of the private members are initialized.

**virtual** ~Protester();

Does nothing since no dynamic memory here.

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

doSomething() is a very long and thorough function that is detailed in the spec for the protester. Since I was unable to implement the tracking feature that the Hardcore protester has, doSomething() is the same for both regular and Hardcore protesters. There are several main things that a protester can do in this: 1. Get stunned, 2. Shout at TunnelMan, 3. Move towards TunnelMan if he is in the line of sight, 4. Move perpendicularly if at a junction. All of these were implemented with helper functions detailed below and the ticks between each action (i.e. non-resting ticks between shouts or ticks between perpendicular movements).

//Accessor and mutator functions common to all types of Protesters

**bool** getLeave();

Returns a value indicating if the protester is in a leave state or not.

**void** setLeave();

Sets leave state to true.

//Functions used by doSomething

**void** moveInDirection(Direction dir);

This function moves one step in a direction that is entered. I utilized a simple switch function alongside the moveTo function.

**bool** inShoutingRange(**int** curX, **int** curY);

Checks if the protester’s x and y value individually are within 4 units and if it is facing the TunnelMan’s direction. This is done by checking left, up, right, and down for the TunnelMan and returning true if either of these directions return true.

**bool** verticalLOS(**int** curX, **int** curY, Direction& dir);

This function is used to check if the TunnelMan is in the Protester’s vertical line of sight. This is done by first checking if they have the same horizontal value and if they do, the path between the TunnelMan and Protester is checked to make sure it is not obstructed (the pathObstructed function is utilized here). If any obstruction is found, false is returned immediately. If the entire path between the TunnelMan and the protester is clear then the protester starts walking towards the TunnelMan and by passing a Direction variable by reference, the doSomething() can use its updated value to find out which direction the protester needs to move in.

**bool** horizontalLOS(**int** curX, **int** curY, Direction& dir);

Same as vertical but done for a horizontal path.

**void** perpendicularHelper();

This is used to check if the perpendicular paths are free by first looking at the left and right directions and checking the perpendicular paths and then looking up and down and checking for viable perpendicular paths. The pathObstructed function is utilized here once again to determine if a path is free or not.

**virtual** **void** gotBribed();

If the protester has been bribed, this function will be called and it will increase the user’s score by 25 points and it will cause the protester to give up and die (note: since I was unable to implement the exit feature, the protester simply gives up and dies if its hitpoints reach zero or is bribed).

**void** setBribed();

Called by GoldNugget class to set the Boolean bribe value to true for the protester.

**void** setStunned();

Called by gotBribed() or shootProtester() to stun the protester by setting it’s stunned Boolean value (private member) to true.

**bool** isStunned();

Checks if a protester is stunned or not.

**HardCore Protester class: derived from Protester**

HardcoreProtester(StudentWorld\* myWorld);

The constructor uses a member initializer list to initialize the Protester base class by inputting true for hardcore, 20 for health, and the imageID of a hardcore protester.

**virtual** ~HardcoreProtester();

Does nothing.

**virtual** **void** gotBribed();

This gotBribed function is different from the base class since it does not kill the hardcore protester but rather it simply stuns him. It also increases the player’s score by 50 rather than 25.

**Earth class: derived from Actor**

Earth(**int** startX, **int** startY, StudentWorld\* myWorld);

It simply sets visible to true. The initialization for an Earth object is done as a 2D array is created in init().

**virtual** ~Earth();

//Added to prevent it from being an ABC

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

Earth cannot do anything.

**Boulder class: derived from Actor**

Boulder(**int** startX, **int** startY, StudentWorld\* myWorld);

Every Boulder contains an enum member called BoulderState which can be either stable, waiting, or falling. The constructor initializes the state as stable.

**virtual** ~Boulder();

Does nothing.

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

If the Boulder is dead, return immediately.

Otherwise use a switch statement with 3 cases: stable, waiting, and falling.

In the stable case, the checkBelowBoulder function from StudentWorld is called and if it returns false, change state to waiting and the countdown is started.

In the waiting case, if 30 ticks have passed transition into the falling state and play the falling sound.

In the falling case, if the bottom is reached set to dead, otherwise if a collisionTunnelMan returns true, we kill the TunnelMan, otherwise if there is another boulder below we set current boulder to dead. We then check for Protesters nearby and kill them if they are in a 3 unit radius. Otherwise, continue falling down.

**Squirt class: derived from Actor**

Squirt(**int** startX, **int** startY, Direction dir, StudentWorld\* myWorld);

Initialize the private member travelDistance to 4.

**virtual** ~Squirt();

Does nothing.

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

If the travelDistance is zero we kill the Squirt object using setDead().

If we are near a protester, we invoke shootProtester to stun it.

The travel distance is then decremented.

A switch statement is used with the input being the current direction and each case is a different direction which is checked using pathObstructed and then moved if there is no obstruction. If in any of these cases the path is obstructed one step in the current direction, the Squirt is set to dead.

**Goodie class: derived from Actor**

Goodie(**int** imageID, **int** startX, **int** startY, StudentWorld\* myWorld, ObjectType type);

Initializes ticks elapsed to be zero.

**virtual** ~Goodie();

Does nothing.

**int** getTicksElapsed();

Returns the number of ticks that have elapsed since goodie was spawned.

**void** incTicks();

void Goodie::setVisibility(bool set){ visibility = set; }

Sets visibility value to true

bool Goodie::getVisibility(){ return visibility; }

Returns whether or not a goodie is visible (replacement for isVisible() function).

**GoldNugget class: derived from Goodie**

GoldNugget(**int** startX, **int** startY, **bool** dropped, StudentWorld\* myWorld);

If the dropped bool value is false, the visibility is set to false (since it must be invisible when spawned). Otherwise, it it set to true.

**virtual** ~GoldNugget();

Does nothing.

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

If the nugget is invisible and in close proximity to TunnelMan, the visibility is set to true using the setVisibility() function.

Else if the gold nugget is not dropped and is within a 3 unit radius of TunnelMan, it is set to dead and the number of gold nuggets the TunnelMan is incremented by 1, a sound is also played.

Else if the gold nugget has been dropped, the bribeProtester function is called which is implemented in StudentWorld and it stuns/kills a protester. If this dropped nugget has been alive for 100 ticks (obtained using getTicksElapsed), we set it to dead.

**SonarKit class: derived from Goodie**

SonarKit(**int** startX, **int** startY, StudentWorld\* myWorld);

Sets sonar kit to visible

**virtual** ~SonarKit();

Does nothing

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

If it is not alive, return immediately.

If it is within a 3 unit distance of the TunnelMan, the SonarKit is set to dead and the sound of acquiring a goodie is placed. The sonarkits the Tunnelman has is also incremented.

If the lifetime value (computed via max(100, 300 - 10\*(getWorld()->getLevel())) ) is reached, the SonarKit is set to dead (i.e. invisible).

**WaterPool class: derived from Goodie**

WaterPool(**int** startX, **int** startY, StudentWorld\* myWorld);

Sets water pool to visible

**virtual** ~WaterPool();

Does nothing

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

If it is not alive, return immediately.

If it is within a 3 unit distance of the TunnelMan, the WaterPool is set to dead and the sound of acquiring a goodie is placed and score is increased by required amount. The squirts the Tunnelman has is also incremented.

If the lifetime value (computed via max(100, 300 - 10\*(getWorld()->getLevel())) ) is reached, the WaterPool is set to dead (i.e. invisible).

**Barrel class: derived from Goodie**

Barrel(**int** startX, **int** startY, StudentWorld\* myWorld);

Initially spawned as invisible.

**virtual** ~Barrel();

Does nothing.

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

If it is not alive, return immediately.

If it is within a 4 unit distance of the TunnelMan and getVisibility is false, the Barrel invokes setVisibility().

Else if it is within a 3 unit distance of the TunnelMan, increase the score by the required amount, decrement number of barrels remaining and play the barrel found sound.

1. A list of all functionality that you failed to finish as well as known bugs in your classes

Due to the time constraint, I was unable to finish two main functionalities in the Protester classes: the walk to exit feature which a Protester once it is in the leaveTheOilField state and the cellphone feature of the HardCore protester which allows it to track the TunnelMan. While I was able to implement a maze queue maze solving algorithm, I could not figure out a way to create a path from the current position to the end points of the maze.

1. A list of design assumptions made

I did not make any notable design assumptions since I felt Piazza, talking to peers, and using the sample removed any ambiguity about the spec that I had. I was only unsure about how the TunnelMan was able to get so close to the Boulder in the sample meanwhile mine was not so I just assumed that the TunnelMan can get at best 3.0 radii units close to the Boulder and since it was not specified where to measure the radius from, I decided to measure it via the center of the objects. I was also unsure about what line of sight meant so I decided to implement it as when the Protester is at the same horizontal OR vertical position as the TunnelMan and there is a clear vertical OR horizontal path between them

1. A description of how you tested each of your classes

**StudentWorld class:**

This class was tested simply by starting and restarting the game to ensure there are no bugs in init() or cleanUp(). After the TunnelMan and digging feature was fully implemented along with Barrels of oil, the TunnelMan was used to collect barrels of oil until it collected all of it and moved onto the next level. If the game did not move onto the nextLevel(), did not spawn the correct amount of objects, or did not display the correct game stat text, there was a problem in init, move or cleanUp. Since the core of StudentWorld is the move() function, testing this out generally involved testing each object individually to ensure its doSomething() operated correctly.

**TunnelMan class:**

Made sure that it moved correctly based on user input and did not run out of the screen. Made sure it dug the Earth and did not get too close to a Boulder. Once I implemented all of the goodies, I made sure the TunnelMan correctly dropped/shot/used its goodies once the keys were inputted. Testing this generally tested each individual goodie itself since their doSomething() functions would be invoked. Furthermore, every time a goodie was used, I checked the game stat line to ensure it correctly incremented/decremented the values. I also tested the TunnelMan by purposely going under a boulder and ensuring it correctly died.

**Boulder class:**

The Boulder class was tested by making sure it killed the TunnelMan, the protester, and correctly fell when there was no Earth present below it. This was fairly trivial to test.

**Squirt class:**

The Squirt class was tested by shooting at Boulders and Earth to make sure no squirt came out and it was correctly decremented. It was then shot in clear paths to make sure it was being created and destroyed correctly and was travelling 4 squares. Once the protester class was implemented, the squirt invoked shootProtester and this was tested by shooting protesters and seeing if they were stunned and eventually left the maze (in my case if they died since I did not implement the mazeLeaving feature).

**GoldNugget class:**

The GoldNugget class was tested by checking if the TunnelMan correctly picked it up. Once this was established, it was used to bribe protesters and this was tested by dropping it right in front of protesters and making sure the correct sound was played, the score was increased by the correct amount, and the protester was stunned.

**Barrel class:**

This was implemented by seeing if it would correctly appear if the TunnelMan went 4.0 units near it and if it was correctly corrected and made the right sound + incremented score correctly when picked up. Furthermore, if all the oil barrels were correct and the level did not change, this would indicate an error.

**SonarKit class:**

This was tested in the TunnelMan class by pressing Z and making sure the items nearby (if any) were correctly revealed. To ensure it also spawned in the correct location, I waited in every level for it to spawn and made sure it was in the top left corner.

**WaterPool class:**

This was a simple class to test, it had to spawn in Earth-less spots and it was tested by seeing if it correctly disappeared after a certain no. of ticks and it disappeared when picked up by the Tunnelman and increased the no. of water correctly.

**Protester class:**

The following features of the protesters were tested:

1. Picking up gold: Ensure the protesters are correctly stunned and the regular protesters exit the oil field whereas hardcore protesters are temporarily stunned.
2. Getting shot: They are stunned when squirted and their health value is decreased. If squirted enough times, they die. Once they give up, ensure TunnelMan’s score increased correctly.
3. Getting hit by boulder: Make sure they die and the boulder continues falling even after bonking the protester.
4. Shouting at TunnelMan: this was tested by walking towards the protesters and making sure they were actually shouting at the TunnelMan. The shoutingTick frequency was increased for testing purposes to see if the shouting correctly decreased the TunnelMan’s scores.
5. Checking for perpendicular paths: This was tested by decreasing perpTicks from 200 to 10 and digging perpendicular paths to see if the Protester would correctly follow it.
6. Seeing if Protester follows TunnelMan when it is in line of sight: This was tested by standing at the same vertical/horizontal position but at a distance and seeing if the Protester followed when there was a clear path between the TunnelMan and protester.