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.”

class Actor : public GraphObject {
public:
    Actor(StudentWorld* world, int imageID, int startX, int startY, Direction dir, double size, unsigned int depth);
This initializes the Actor and sets it as visible. It is not virtual as it is a constructor for the Actor class.

    virtual ~Actor();
This hides the Actor as it is destroyed.

    StudentWorld* getWorld();
This function returns a pointer to StudentWorld that is used to create the game. I decided not to make it virtual as it doesn’t need special functionality.

    bool isAlive();
This function returns the status of whether or not an actor is alive. It is not virtual is it doesn’t need special functionality.

    void die();
This function sets an actor's life status to false. It is not virtual as it doesn’t need special functionality.

    void moveTo(int x, int y);
This function ensures that an actor can only move within bounds, it is not virtual as it does not need special functionality.

    virtual void doSomething();
This function allows each class to be able to do something for each tick. It is virtual as each actor has their own special version of the function.
};

class Earth : public Actor {
public:
    Earth(StudentWorld* world, int startX, int startY);
This constructs the earth object at each location. It’s the constructor for the Earth class.

    virtual void doSomething();
This function allows each class to be able to do something for each tick. It is virtual as each actor has their own special version of the function.

};

class Human : public Actor {
public:
    Human(StudentWorld* world, int imageID, int startX, int startY, Direction dir, int hp);
This initializes a human with a given hp.

    int getHp();
This function returns the health of the human actor. It is not virtual as it does not need special functionality.

    void subtractHp(int pts);
This function subtracts a certain amount from a human's health. It is not virtual as it does not need special functionality.

    virtual void doSomething();
This function allows each class to be able to do something for each tick. It is virtual as each actor has their own special version of the function.

    virtual void moveInDir(Direction dir) = 0;
This is a pure virtual function that allows a human to move in a direction. Each class has different ways of movement so it is pure virtual.

};

class Tunnelman : public Human {
public:
    Tunnelman(StudentWorld* world);
This constructs a Tunnelman with it's initial attributes.

    virtual void doSomething();
This function allows each class to be able to do something for each tick. It is virtual as each actor has their own special version of the function. The Tunnelman, specifically, is able to move through inputs by the user. The Tunnelman is able to move, use water, use gold, or use sonar.

    virtual void isAnnoyed(int hp);
This function subtracts health from the Tunnelman when it is annoyed, when their health is below 0, the tunnelman dies. It is virtual as other Humans can also be annoyed.

virtual void moveInDir(Direction dir);
This function allows the tunnelman to move in a direction. It is pure virtual as each class has different ways of moving.

void addItem(int itemId);
This function allows the tunnelman to add to their stores of water, gold, or sonar. It is unique to the tunnelman so it is not virtual.
int getWater();
This function returns how much water the tunnelman has.

int getSonar();
This function returns how much sonar the tunnelman has.

int getGold();
This function returns how much gold the tunnelman has.

void shootWater();
This function allows the tunnelman to shoot water squirts.

};

class Boulder : public Actor {
public:
Boulder(StudentWorld* world, int startX, int startY);
This constructs a Boulder with it's initial attributes and checks if it can be placed onto the field.

virtual void doSomething();
This function allows the boulder to check for nearby humans to annoy and it handles the behavior of the boulder when it is falling. It is a virtual function as each class has different functionality.

void annoyHuman();
This function damages any nearby humans within it's radius. It's unique to the Boulder class so it's not virtual.

class Squirt : public Actor {
public:
Squirt(StudentWorld* world, int startX, int startY, Direction dir);
This function constructs a squirt with it's attributes.

virtual void doSomething();
This function moves the squirt in it's direction and checks for collisions or when it should be removed. It's virtual because each class has it's own functionality

    bool annoyProtesters();
This function damages any protesters in it’s radius. It’s not virtual because it’s unique to the squirt class.

};

class Item : public Actor {
public:
    Item(StudentWorld* world, int imageID, int startX, int startY);
This function initializes an Item with it’s attributes.

    virtual void doSomething() = 0;
This function allows each item to do something. It’s pure virtual as each item has it’s own functionality.

    virtual void lifetime(int time);
This handles the lifespan of an item and it determines when an item should be removed. It’s virtual as each item has a different lifespan.

};

class OilBarrel : public Item {
public:
    OilBarrel(StudentWorld* world, int startX, int startY);
This initializes an oil barrel and hides it.

    virtual void doSomething();
This makes the barrel visible when the player is near and handles its destruction when it is collected. It’s virtual as each class has a different functionality.

};

class GoldNugget : public Item {
public:
    GoldNugget(StudentWorld* world, int startX, int startY, bool isVisible, bool isPickupable);
This constructs a gold nugget that handles its visibility and whether or not it can be picked up.

    virtual void doSomething();
This makes the nugget visible when it’s near a player and handles it’s collection and interaction with protesters. It’s virtual as every class has a different functionality.

};

class SonarKit : public Item {

public:
    SonarKit(StudentWorld* world, int startX, int startY);
This constructs a sonar kit with a lifespan based on the game level.

    virtual void doSomething();
This handles the collection of the kit by the player and its lifespan. It's virtual as each class has a different functionality.

};

class WaterPool : public Item {
public:
    WaterPool(StudentWorld* world, int startX, int startY);
Constructs a WaterPool with a variable lifespan based on the game level.

    virtual void doSomething();
  Manages the WaterPool's collection by the player and handles its lifespan.

};

class Protester : public Human {
Public:

    Protester(StudentWorld* world, int imageID,int hp);
Initializes a Protester with specified attributes and rest periods.

    virtual void doSomething();
Handles the Protester's behavior, including movement, interaction with the player, and changing direction.

    void bribeProtester();
Processes the bribery action, affecting the Protester's behavior based on whether or not it's a regular protester or a hardcore one.

    void moveInDir(Direction dir);
This moves the Protester in a specified direction and handles boundary conditions.

    virtual void isAnnoyed(int hp);
This processes damage taken by the Protester and handles its reaction and potential death.

    void stunProtester();
This function resets a protester's rest period, allowing them to pause before moving again.

```
    bool isFacingPlayer();
```
This function checks if the protestor is facing the player based on it's direction

```
    Direction dirToPlayer();
```
This function determines the direction towards the player from the Protester's current position.

```
    bool isTherePathToPlayer(Direction dir);
```
This checks if there's a clear path to the player based on the direction.

```
    bool canTurn();
```
This determines if a protestor can turn based on it's location and surroundings

```
    void pickDir();
```
This chooses a direction for a protestor to move based on the surroundings

```
    Direction randomDir();
```
This generates a random direction for the protestor to move

```
};

class RegularProtester : public Protester {
public:
    RegularProtester(StudentWorld* world);
```
This initializes a regular protestor with it's given attributes
```
};

class HardcoreProtester : public Protester {
public:
    HardcoreProtester(StudentWorld* world);
```
This initializes a hardcore protestor with it's given attributes.
```
};

#endif // ACTOR_H_

class StudentWorld : public GameWorld
{
public:
    StudentWorld(std::string assetDir);
```
This initializes a new StudentWorld with a specified asset directory.

```
    ~StudentWorld();
```
This cleans up any dynamically allocated memory for the actors, the earth, and the player.

virtual int init();
This virtual function initializes the game world, sets up the oil field, and places all players and objects

virtual int move()
This updates the state of the game by processing all actors and checking for game-end conditions.;

virtual void cleanUp();
This functions like the destructor and deletes all dynamically allocated memory for earth, actors, and player, cleaning up the game world.

bool canDig(int x, int y);
This checks and removes earth blocks in a 4x4 region centered at (x, y) if present.

void updateText();
This updates the game's status display with the current score, level, lives, health, water, gold, sonar, and barrels.

std::string formatText(int score, int level, int lives, int hp, int water, int gold, int sonar, int barrels);
This formats and returns a string with the game's current statistics.

void addActor(Actor* actor);
This adds a new actor to the list of actors in the game world.

void addItem();
This randomly adds a new item (Sonar Kit or Water Pool) to the game world based on certain conditions.

void addProtestor();
This adds a new protestor (Regular or Hardcore) to the game world based on the level and other conditions.

void removeProtestor();
This decreases the count of protesters in the game.

void removeBarrel();
This decreases the count of barrels in the game.

bool isInRadius(int x1, int y1, int x2, int y2, int radius);
This checks if the points x2 and y2 are within the radius of x1 and y1.

bool isActorInRadius(int x, int y, int radius);
This checks whether an actor is within the specified radius of x and y.

void addObject(int num, std::string type);
This allows an object (barrel, gold, boulder) to be added to the oil field.

bool isAboveEarth(int x, int y);
This checks for blocks above the specified area of the field

bool isThereEarth(int x, int y);
This checks whether there is an earth block within the specified radius

bool isThereBoulder(int x, int y, int radius);
This checks for whether there is a boulder within the specified radius.

bool isPlayerInRadius(Actor* actor, int radius);
This checks for whether a player is within the radius of an actor.

void removeDead();
This function removes any dead actors or objects from the field

void dropGold();
This function drops gold nuggets from the player's location

void useSonar();
This function reveals any barrel or gold within the player's radius.

bool canMoveInDir(int x, int y, GraphObject::Direction dir);
Determines if a move in the specified direction from (x, y) is possible based on obstacles.

Protester* isProtesterInRadius(Actor* actor, int radius);
This function checks if there is a protester within radius of an specified actor.

Tunnelman* getPlayer();
Returns a pointer to the player (Tunnelman) in the game world.

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I wasn't able to implement the Squirt class." or "My Hardcore Protester doesn't work correctly yet so I just treat it like a Regular Protester right now."

I was not able to implement the functionality of Protesters exiting. I was not exactly sure how to go about letting the Protestors be able to exit, and that was the main problem of my implementation of the project.

3. A list of other design decisions and assumptions you made, e.g.:
i. It was ambiguous what to do in situation X, and this is what I decided
to do.

I was not sure how to format the update text as the image on the pdf and the instructions did not have the same format. I decided to go with the way the instructions had it, which was with level first and score last.

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

StudentWorld Class
The StudentWorld class was tested by checking whether or not it could initialize and manage the game world correctly. I checked if the earth, actors, and player were correctly initialized and if they performed as expected. It was also tested to see if boulders, gold, and oil barrels were created and were in valid positions. Movement and interactions were tested to see if their statuses were updated.

Actor Class
The actor class was tested by verifying that actors were positioned correctly, could perform actions as expected, and interacted properly with other game elements. Because it was the base class for all the other classes, the derived classes were also checked for their functionality.

Tunnelman Class
I tested the Tunnelman class by moving, digging, and using tools. I checked that the player's movement was restricted correctly by game boundaries and obstacles, that the digging cleared earth properly, and that inventory was correctly managed. Interactions with other game elements, such as protestors and collectibles, were tested to ensure that the Tunnelman could correctly pick up items and perform necessary actions.

Protester Class
I interacted with hardcore and regular protestors. I verified their movement patterns, reactions to the player, and ability to challenge the player according to their difficulty level. I also checked with how they responded to players and objects in the field/

Boulder Class
I checkled that boulders could be placed correctly and that they blocked movement as intended. I also checked it's falling mechanism and it's interactions with the player and the protestors.

GoldNugget Class
I checked the correct placement and collection of gold nuggets. I ensuensuredring that gold nuggets appeared in valid locations and could be collected by the player. I also checjed when dropped by the player, ensuring that it could be picked up again or interacted with appropriately.

OilBarrel Class
I checked that oil barrels were placed correctly in the game world and that their collection by the player functioned as intended. I verified that the number of remaining barrels was updated correctly and that collecting all barrels triggered the level completion status.

SonarKit Class
I verified that the sonar functionality correctly revealed hidden items and actors within the specified radius. The effect of the sonar kit on the game world and its interaction with other game elements were also tested to ensure correct behavior.

WaterPool Class
I checked that water pools appeared in the game world and that they could be collected by the player. I verified that the water supply of the player was updated correctly upon collection and that the water pools were placed in valid locations.