Report

**1. High Level Description:**

class Actor

Actor(int imageID, int startX, int startY, Direction startDirection,

float size, unsigned int depth, StudentWorld\* world);

Actor is a derived class from GraphObject, and thus accepts parameters to initialize the GraphObject variables. In addition, it accepts a pointer back to StudentWorld, and a bool variable that keeps track of whether an actor is alive or not.

virtual ~Actor();

The destructor sets every dead actor to invisible, and also their m\_isAlive state to false.

virtual void doSomething() = 0;

This function is used by all of the objects, and is never needed to be implanted, and thus, it was made pure virtual. Moreover, actor is an abstract base class, and would never need to be created because an actor can't do anything, only it's derived classes.

virtual void getAnnoyed(int howMuch) = 0;

The actor class is an abstract base class, and thus it will never getAnnoyed, only it's derived classes will. I chose to define this function as pure virtual because all actors of the game must use this function or implement it in some way. It takes one parameter, and some classes to change their annoyance points.

virtual bool canActorsPassThroughMe() const;

This function was made virtual because some objects can just inherit that they can be passed through, while others cannot be passed through, and thus will set this to the opposite boolean value.

bool isAlive() const;

Every function will use this method in the exact same way as the Actor class, and so it was neither virtual nor pure virtual. The function merely returns the value of the m\_isAlive variable depending on its status.

void setAlive(bool value);

This function is used by all classes, and never needs to be changed, as it merely sets the state of a given actor to alive or dead.

bool moveDir(int direction, int numSquaresToMove);

This function accepts two parameters: one is the direction that the frackman is facing, and the other is the number of squares to move in a given direction. It returns a boolean value of whether or not it can move in a given direction, and then moves the object if it can move.

StudentWorld\* returnWorld() const;

This function returns the StudentWorld pointer back to the StudentWorld class.

virtual void setBribe() = 0;

This function is used to set the bribe (gold) of the protester class. It was used as a pure virtual function because this Actor class never needed to implement it, and the protesters would use it in a different way. Because the protesters were in a vector, and these classes didn't have pointers to protesters, I put this function in the base class because StudentWorld had a vector of actor pointers.

class Protester

Protester(int imageID, int startX, int startY, Direction startDirection, float size, unsigned int depth, StudentWorld\* world, int health);

This class is derived from the actor class, and thus all, but one of its parameters is used to initialize the actor class. The only exception to the constructor is a health member variable that was added to the class. In addition, the protester always starts out as visible.

virtual void doSomething() = 0;

This function was chosen to be pure virtual because each of the protestors (regular protestor and hardcore protestor) will implement this in different ways. Because both protestors will use different methods of the function, it can be pure virtual.

virtual void doesTheSomething();

This function is used to take care of what the protester must do each tick of the game. It first checks to see if the protester is alive, and if not, the function instantly returns, to delete the dead protester from the game. It then checks to see if the protester is in a leave the oil field state, and if so, the protester will set its resting ticks to zero. It then must check its resting ticks and new resting ticks to wait the proper amount of ticks before the protester is able to do something again. Then it checks to see if is in a leave the oil field state, and if so will take the proper protocol to finds its way to the exit and leave the field. If the protester is not in a leave the oil field state, then the protester checks to see if it is within a striking distance of the frackman. If so, then the protester yells at the frackman, properly annoys the frackman (with the correct amount of annoyance points), resets the ticks to wait before another shout can be used, and resets the ticks to wait before another move can be made. It then immediately returns. Also there must be a check to see if the protester is right next to the frackman, and if so, the protester shouldn't move, but wait until the next opportunity to yell at the frackman. The next action checks to see if the protester is in a direct line of sight of the frackman, but is at a distance of more than four steps away from him. It then will continue to move towards the frackman, given that it's path is not obstructed. If it can move, then it will continue to make steps towards the frackman. If a protester has finished its given number of steps for a given tick, and can no longer move in a given direction, then the protester will change its direction to a valid position to move to.

virtual void getAnnoyed(int howMuch);

This function accepts one parameter, which is the amount of how much the protester should be annoyed by. This function is used to handle what the protester should do when it is annoyed. First the annoyance points are decreased from the protester's health points, and then the protester checks to see if its health has been completely diminished. If so, then the protester will set its state to leave the oil field, play the protester give up sound, set it's resting ticks to zero, and increase the player's score. If the protester got annoyed, but was not killed, then the annoyed protester sound is played, and the resting ticks are reset for the next move.

virtual void setBribe();

This function handles what should be done when the protester picks up a piece of gold. If the protester comes into contact with gold (bribe), then the protester will increase its gold variable by one and sets its leave the oil field state to true. The regular protester should prepare to leave the oil field, when it picks up a piece of gold.

int getHealth() const;

This function simply returns the health of a protester back to its caller. It didn't need to be a virtual or pure virtual function, because there would never be a case where a protester would use this function in a different manner.

void decHealth(int hitpoints);

This function accepts one parameter, which is the amount of points that is to be decreased from the protester's health points. It didn't need to be virtual or pure virtual because similarly to the getHealth() function, no protester would ever use a different instance of this function.

void setDead();

This function is used to set the state of the protester to dead. It checks to see if the caller has provided the right ID for either the protester or the hardcore protester, and then plays the sound, for a protester giving up. In addition, it sets the health to zero, as the protester was killed, and sets both the visibility state and alive states to false.

void changeDirectionToMove();

This function is used to change the direction of the regular protester if the protester has run out of steps to move and needs to pick a new direction. It will pick a random direction, and will check to see if the new direction, is a valid direction to take a step. If not, it will continue checking other directions to take a valid step. It will then set its direction to the valid direction, and reset the number of steps to take.

bool canMove(int x, int y, int direction);

This function accepts three parameters that are the coordinates of a given object and the direction that the object is facing. It returns a boolean value, and checks to see if the object can move in a given direction. It returns true if so, and false otherwise.

void doMove(int x, int y, int direction);

This function accepts three parameters that are the coordinates of a given object and the direction that the object is facing. It is used to move the object if it is possible. It runs through the four possible directions and calls the canMove function to see if it can move in a given direction, and if so, moves in that direction.

class RegularProtester:

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

The regular protester class is a derived class from the protester class, and thus it initializes a protester with given values. This constructor accepts three parameters, which are the starting coordinates, as well as a pointer back to the StudentWorld class. A regular protester has four variables in the body of the constructor. The first is the number of moves that a regular protester can move each turn. Another is the number of ticks that the regular protester must rest, while the waitingTicksToShout variable is used to keep track of the last time that the protester yelled at the frackman. Finally, there is a boolean variable that is to keep track of whether the regular protester has been fully annoyed or bribed by gold, and must leave the oil field.

virtual void doSomething()

All this function does is call the doesTheSomething() function of the protestor base class.

class HardcoreProtester

virtual void doSomething();

All this function does is call the doesTheSomething() function of the protestor base class

virtual void setBribe();

This function is virtual because unlike the regular protestor, the hardcore protestor doesn't leave the oil field when it picks up gold. Therefore, it needed to be implemented differently.

class FrackMan

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

The frackman class is derived from the actor class, and properly initializes the actor portion of the class. The frackman class also has four variables that keep track of the squirts, sonars, gold, and health points. These are properly initialized, and the visibility of the frackman is set to true.

virtual void doSomething();

This function handles every tick of the frackman. First, the frackman checks to see if it is alive, and if not, the function instantly returns. It then checks to see if a valid key was input by the user, and if so, enters into a switch statement to properly handle each key. If a direction key was entered, then the function will check to see if there is any thing obstructing the frackman from moving in the given direction. If the frackman can move in that direction, then the frackman will move there, if there is dirt in the move, then the dirt will be removed, and the dig sound will be played. If the frackman is at the edge of the map, then it will move in a stationary form like the game provided does. If the user pressed the space bar, then the function will take care of using a squirt object. If the squirt count is greater than or equal to one, then a new squirt object will be added to the oil field given by the direction and coordinates of the frackman. It will then play the squirt sound. If the user pressed the tab key, then the frackman will drop a gold (bribe), which will be added to the oil field. If the user pressed the escape key, then the frackman's state will be instantly set to dead, and will restart the level (given the frackman still has lives left). If the user pressed the z key, then the function will handle the use of the sonar. If the frackman is within a radius of 12 squares from any goodie (that can be activated), then their visibility will be set to true. Finally the sonar sound will play.

virtual void getAnnoyed(int howMuch);

This function accepets one parameter, which is the amount of how much to annoy the frackman with. Each time the frackman gets annoyed (by a protestor), then 2 points get decreased from the health points. If the frackman's health reaches zero, then the frackman's state gets set to dead, and the player give up sound is played.

bool validPosition(int x, int y) const;

This function checks the initial coordinates of the frackman to make sure that he is in a valid location of the oil field. Returns false if not.

int getSquirts() const;

This function returns the number of squirts that the frackman has left.

int getSonars() const;

This function returns the number of sonars that the frackman has left.

int getGold() const;

This function returns the number of gold that the frackman has left.

int getHealth() const;

This function returns the amount of health that the frackman has left.

void setSquirts(int howMuch);

This function is used to set the squirts of the frackman, when he picks up water.

void setSonars(int howMuch);

This function is used to set the sonars of the frackman, when he picks up a sonar object.

void setGold(int howMuch);

This function is used to set the gold of the frackman, when he picks up a gold object.

void setDead();

This function is used to set the state of the frackman to dead, when he had been killed (by a boulder). It will play the player give up sound, set the visibility and alive states to false, and set the health the zero.

virtual void setBribe();

This function is not used by the frackman

class Dirt

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

This class is derived from the actor class, and thus initializes the actor variables properly. In addition, dirt starts out visible, so the visibility is set to true.

virtual void doSomething();

Dirt doesn't do anything, so left empty.

virtual void getAnnoyed(int howMuch);

Dirt doesn't get annoyed, so left empty.

virtual bool canActorsPassThroughMe() const;

Actors can't pass through dirt (only mined by frackman), so state is set to false.

virtual void setBribe();

Dirt doesn't get bribed.

class Boulder

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

This class is derived from the actor class, and thus initializes the variables properly. The boulder also has two new variables to keep track of the state of the boulder, as well as a counter to keep track of the waiting state. In addition, a boulder starts out visible, so its state is set to visible.

virtual void doSomething();

This function handles what a boulder does each tick of the game. It first checks to see if the boulder is alive, and if not the function instantly returns. It then checks the state of the boulder. If the boulder is in a stable state (there is dirt below), then the boulder does nothing. If dirt was removed from below the boulder, then the boulder will enter a waiting state, and start to increment the counter for the waiting state. It then checks to see if the boulder is in a waiting state. If so, then the boulder will check to see if its waiting counter has reached 30, if so, then the boulder will enter a falling state. If it is in a falling state, then the falling rock sound will play. If the boulder's waiting counter is less than 30, then increment the waiting counter. It then checks to see if the boulder is in a falling state, if so then the boulder will call the moveBoulder() function to properly handle the falling process.

virtual void getAnnoyed(int howMuch);

Boulders do not get annoyed, so empty.

bool isDirtBelow() const;

This function returns a boolean value, and checks to see if there is currently dirt below the boulder. If there is dirt below, then the function returns true, and if not false.

int getWaitingStateCounter() const;

Returns the current waiting counter number back to its caller.

void moveBoulder();

This function handles what to do if the boulder is in a falling state. It must check three things: if there is a frackman below, if there is a protestor below, or another boulder below. If there is nothing below, then each tick, it will move one step down. If the boulder comes into contact with a frackman, then the boulder will call the killFrackMan() function to kill the frackman. If there is a protestor below, then the function will properly annoy the protestor, and will increase the player's score by 500. If there is another boulder below or dirt below, then the function will set the boulder's alive state to false because the boulder can't move anymore.

virtual bool canActorsPassThroughMe() const;

Boulders cannot be passed through, so return false.

virtual void setBribe();

Boulders can't be bribed, so left empty.

class Barrel

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

The barrel class is derived from the actor class and thus initializes the variables of the actor variables. Barrels have no other variables, and start out invisible, so their state inherits the set alive invisible state.

virtual void doSomething();

This function handles what a barrel should do each tick. It first checks to see if the barrel is alive, and if not the function instantly returns. It then checks to see if the barrel comes within a radius of four of the frackman, and if so, then the barrel will set its state to visible to be seen by the frackman. It then will check to see if the frackman is within a radius of 3 from the barrel, and if so, then the barrel is picked up by the frackman. The barrel gets set to dead, and the barrel variable in the StudentWorld class is properly reduced. The found oil sound is played, and the score of the player is increased by 1000.

virtual void getAnnoyed(int howMuch);

Barrels don't get annoyed, so left empty.

virtual void setBribe();

Barrels don't get bribed, so left empty.

class Gold

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

This class is derived from the actor class, and thus initializes the variables of the actor class properly. Gold starts out invisible, so it inherits the invisibility state from the actor class. Gold has no other variables.

virtual void doSomething();

This function handles what gold should do each tick. If the gold is dead, then the function instantly returns. If gold is invisible, and the frackman is within a radius of 4 of the gold, then the visibility gets set to true and the function returns. It then checks to see if the frackman is within a radius of 3 of the gold object. If so, then the gold is picked up by the frackman, the gold gets set to dead, the got goodie sound is played, the score of the player is increased by 10, and the gold is properly updated for the frackman.

virtual void getAnnoyed(int howMuch);

Gold doesn't get annoyed, so left empty.

virtual void setBribe();

Gold can't be bribed, so left empty.

class Bribe

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

This class is derived from the actor class, and thus properly initializes the variables of the actor class. In addition, the bribe class has a variable that keeps track of the ticks that the bribe should be left on the oil field before dying. In addition, bribe starts out in a visible state (because the frackman drops the bribe).

virtual void doSomething();

This function handles what a bribe should do each tick. As every tick of the game passes, m\_ticks gets decremented by 1. The function then checks to see if a protestor is within a radius of 3 of the bribe object. If so, then the protestor will pick up the bribe, the protestor found gold sound will play, the player's score will be incremented by 25, and the function will call the setBribe() function to be properly handled by the protestor class. It then checks to see if the bribe's ticks are less than or equal to zero, and if so, the bribe is set to dead to be removed from the oil field.

virtual void getAnnoyed(int howMuch);

Bribe can't get annoyed, so left empty.

virtual void setBribe();

Bribe can't be bribed (ha), so left empty.

class Sonar

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

This class is derived from the actor class, and thus properly initializes the variables of the actor class. The sonar class also has a tick variable to properly handle how long the sonar should be left in the oil field before disappearing. Also, sonar always starts out visible.

virtual void doSomething();

This function handles what a sonar should do each tick of the game. Every turn one tick is removed from the tick variable. The function then checks to see if the sonar is alive, and if not the function returns. The function then checks to see if the frackman is within a radius of 3 of the sonar object, and if so, then the state of the sonar gets set to dead, the got goodie sound is played, the player's score was increased by 75, and the sonars are increased for the frackman class.

virtual void getAnnoyed(int howMuch);

Sonars can't get annoyed, so left empty.

virtual void setBribe();

Sonars can't be bribed, so left empty.

class WaterSquirt

WaterSquirt(int x, int y, Direction direction, StudentWorld\* world);

This class is derived from the actor class, and thus properly initializes the variables of the actor class. In addition, the squirt object always starts out in a visible state. Squirt objects have a new variable that keeps track of the distance that the squirt should travel.

virtual void doSomething();

This function handles what a squirt object should do each tick. It first checks to see if the squirt object is alive, and if not the function returns. If then checks to see if a protestor is within a radius of 3 of the squirt object, and if so, the function will properly annoy the protestor, set its state to dead, and the visibility to false.

It then will check to see if the distance traveled was at its max, and if so, the squirt object is set to dead. The function then enters a switch statement based off of the direction that the frackman (the squirt object too) is facing, and checks to see if there is dirt in the way. If there is dirt in the way, the squirt object will die. If not, and there is no boulder in the way, then the squirt object will move 1 square in the given direction.

virtual void getAnnoyed(int howMuch);

Squirts can't get annoyed, so left empty.

virtual void setBribe();

Squirts can't get bribed, so left empty.

class WaterPool

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

This class is derived from the actor class, and thus properly initializes the variables of the actor class. This class also has a variable to keep track of how long to remain in the oil field before being removed from the game. In addition, it always starts out in a visible state.

virtual void doSomething();

This function handles what a water pool should do each tick of the game. Every turn, a value of one is decremented from the tick variable. The function then checks to see if the WaterPool is alive, and if not, the function instantly returns. If the ticks is less than or equal to zero, then the WaterPool gets set to dead to be removed from the game. It then checks to see if the frackman is within a radius of 3 of the WaterPool, and if so the WaterPool is set to dead, the got goodie sound is played, the squirt variable of the frackman class is incremented by 5, and the score of the player is increased by 100.

virtual void getAnnoyed(int howMuch);

WaterPools can't be annoyed, so left empty.

virtual void setBribe();

WaterPools can't be bribed, so left empty.

class StudentWorld

virtual int init();

Initializes each level of the game by inserting a frackman, creating the dirt, and adding the initial actors (gold, boulder, oil). When it is complete initializing everything, then it returns to get the game started.

virtual int move();

This function is what composes the movement of the game. It begins by incremented the tick for the given turn and then updates the display text at the top of the screen. It then gives each object a change to do something each tick. If the frackman died, then the level resets or if the frackman is out of lives, the game ends. Also each turn, newly-dead actors are removed after each tick of the game. If the correct number of barrels had been collected for the given level, then the game advances to the next level. Moreover, new actors are added at given ticks of the game (protesters, sonar, water pool etc.)

virtual void cleanUp();

The cleanup function is used to remove all objects from the game due to a level being beaten, a player being killed, or a level ending.

bool isDirt(int x, int y) const;

This function accepts two parameters (the coordinates), returns a boolean value, and checks to see if there is dirt present or not at a stationary position. If the dirt is a nullptr or if it was set to invisible (due to being removed by the frackman), then the function returns true.

bool isDirt(int x, int y, int direction) const;

This function accepts three parameters ( the coordinates, and a given direction), return a Boolean value, and checks to see if there is dirt present at a given position based off of what direction an object chooses to move. It uses a switch statement to check the four directions, and also checks in four 1x1 squares because an object that can move (frackman, protester etc) occupies a 4x4 area.

bool isBoulder(int x, int y, int direction) const;

This function accepts three parameters (coordinates, and direction), return a boolean value, and checks to see if there is a boulder present at a given location. The function has to check in a similar way as the isDirt with three parameters function did, as moving objects occupy a 4x4 region.

bool removeDirt(int x, int y, int direction);

This function accepts three parameters, returns a Boolean value, and removes dirt at a given location. It keeps track of when dirt is being mined (and if so sets it's state to invisible). It first checks to see if the frackman spawned in a dirt location, and if so removed all the dirt at his starting point. Upon moving either left, right, up, or down, it checks to see if there is dirt there, and if so, it will go about removing the dirt, and signaling that dirt was removed to play the sound. If dirt was ever mined, the function returns true to play the sound.

void removeDirt(int x, int y);

This function accepts two parameters, returns a Boolean value, and removes dirt in a 4x4 region. It sets each location of dirt removed to be invisible.

void killFrackMan();

This function is called to kill the frackman. It then calls the set dead function to set frackman's state to dead.

void annoyProtester(int howMuch);

The function accepts one parameter, which is the amount to annoy the protester with. It then loops through all of the actors until it finds a protester (regular or hardcore), and then calls the getAnnoyed function to properly annoy the protester with the correct amount of annoyance points.

void annoyFrackMan();

This function calls the getAnnoyed function for frackman, but always annoys frackman with 2 annoyance points. This function handles when a protester annoys the frackman.

void setBarrel(int howMuch);

This function accepts one parameter, which sets the barrel by how much the parameter is.

void setGold(int howMuch);

This function accepts one parameter, which is how much gold to give to the frackman. It updates the frackmans gold variable, when he picks up gold.

void setBribe(int howMuch);

This function accepts one parameter and is called when a protester picks up a piece of gold. It then loops through all of the actors, until it finds a protester and then gives the protester the gold. There is a count, so as to only give the gold object to one protester.

void addBribe(Actor\* a);

When the frackman drops a piece of gold, a bribe object (gold for protesters) is created and added to the oil field. It is then pushed onto the vector as another actor.

void addWaterSquirt(Actor\* a, Actor::Direction direction);

This function accepts two parameters, one that is a pointer to the given actor, and the other, the actor's direction. The function adds a squirt object into the oil field based on the frackman's location and the direction that he is facing. It also checks to see if there is dirt or a boulder in the way, to avoid placing a squirt object in an incorrect location.

void setSonars(int howMuch);

This function accepts one parameter and adds a sonar to the frackman's sonar variable when he comes into contact with a sonar object on the oil field.

void setSquirts(int howMuch);

This function accepts one parameter and adds a squirt object to the frackman's squirt variable when he comes into contact with a squirt object on the oil field.

double radius(int x1, int y1, int x2, int y2) const;

This function accepts four parameters and returns the radius between given objects. It computes this by using the distance formula. This function is used by many objects and is used to check if they come within a certain range of other object.

int randInt(int min, int max);

This function is used to compute a random number, and uses a method provided in project 1 to do so.

bool isNearFrackMan(Actor\* a, int radius) const;

This function accepts a pointer to a given actor, and a given radius to use, and returns a Boolean value. If the object is within the given radius of the frackman then it returns true, as the object is near the frackman, and false otherwise.

bool isNearProtester(Actor\* a, int radius);

This function works in the same way as the frackman function above, but is used to see if an object is within a given radius to a protester. It does this by looping through the actors, until a protester is found, and then returns true, if near a protester, and false otherwise.

bool isNearBoulder(int x, int y, int r);

This function accepts three parameters and returns a Boolean value of whether or not a given object is within a certain radius to a boulder. It does this by looping through the actors, until it finds a boulder, and then checks to see if the object is near the boulder. If so, it returns true, and returns false otherwise.

//void isNearActor(Actor\* a, int r);

//bool facingTowardFrackMan(Actor\* a) const;

GraphObject::Direction lineOfSightToFrackMan(Actor\* a) const;

This function accepts one parameter, which is a pointer to a given actor object. It returns a direction based on if the protester is in a line of sight of the frackman. It loops through the coordinates, and if the frackman and the protester have the same coordinates, then it returns a direction based on the direction that the protester is facing in regards to the frackman.

//GraphObject::Direction DirToFrackMan(Actor\* a) const;

bool canActorMoveTo(Actor\* a, int x, int y) const;

This function accepts three parameters, which are a pointer to an actor, and x and y coordinates. It checks to see if an actor can move to a given position, and if so return true, and false otherwise. If there is dirt or a boulder in the way, or if the given location to move to is out of bounds, then the function returns false.

**2. Functionality not completed**

* I wasn't able to complete the protestor classes. I have some parts of the regular protestor working, but certainly not all of it.
  + Regular protestor can move, and track down the frackman to a certain extent, but gets caught up at certain areas of the oil field
    - I wasn't able to implement the exit oil field aspect as well
  + Hardcore protestor just works like my limited regular protestor does
* I implemented every other aspect of the game to the best of my ability

**3. Design decisions and assumptions**

The spec was pretty descriptive and guided me well in implementing my classes. I wasn't all that sure on how to use the gold (bribe) object with the protestors, so I made a function pure virtual in the actor class, to allow me to use the actor pointers in the StudentWorld class to properly give the protestor the gold. In terms of the protestors, I wasn't all that sure on how to keep track of the ticks to move, so I used three sets of ticks to keep track of the resting ticks, the ticks to wait before another shout, and new resting ticks that are reset when the protestor does a given action.

**4. Tests for each class**

\*For the tests, I mostly just played my version of the game, and checked to see if the cases matched that of the game provided. I also introduced different initializations of each object to check for certain situations as well.

class RegularProtestor

For the regular protestor class, the tests were limited as I wasn’t able to complete a whole lot of the class. I tested what would happen if the frackman was in a direct line of sight of the frackman and was in a given radius. As such, I tested that the protestor should annoy the frackman, and reset its state so that it would have to wait before he could annoy the frackman again. I also tested that if the frackman was in a direct line of sight of the frackman, and the frackman went out of the line of sight, that the protestor should change direction, and head in that direction.

class HardcoreProtestor

For the hardcore protestor class, the tests were the exact same as the regular protestor, as I wasn't able to complete this class, and it is merely a replicate of the regular protestor class.

class FrackMan

For the frackman class, I tested each key that was pressed by the user. I walked around the entire oil field to make sure, that the frackman could travel in a given direction (given it was a valid position to move). I made sure to move to all edges of the oil field to make sure that the frackman couldn't leave the oil field. I also made sure to approach a boulder at every possible position to make sure that the frackman couldn’t pass through a boulder (if it is at a given position). I also made sure to move through the dirt, and made sure that the dirt was being properly removed.

In regards to other object's interactions with the frackman, I tested different aspects of my game. I checked to see what would happen when the frackman came into contact with other objects of the game (i.e. dirt being mined, a boulder hitting the frackman etc.). I also checked how the protestor would interact with the frackman, and what would happen when the frackman's health was reduced. In terms of the keys pressed by the user, I made sure that the objects (i.e. squirt, dropping gold, sonar) did the same things that the game provided did.

class Dirt

For dirt, I made sure that the size of the oil field occupied by dirt (not in the shaft), were identical to that of the game provided. I also checked to see what would happen when boulders were initialized and made sure that the dirt was removed properly, so that the boulder could be visible in a 4x4 region. In addition, I checked to see that the frackman would mine the dirt properly, and that other objects couldn't pass through the dirt.

class Boulder

For the boulder class, I checked to see what would happen when the boulder was initialized into the oil field. I checked that the boulder would properly be initialized (and visible), and had no dirt blocking any portion of the boulder's 4x4 region in the oil field. I then checked to see that objects (frackman, squirt etc.) couldn't pass through the boulder. Moreover, I checked to see what would happen when the dirt was removed from beneath the boulder. I made sure that the boulder behaved in the same way as the game provided, and would properly disappear when it came into contact with frackman, a protestor, and another boulder.

class Barrel

For the barrel class, I tested what would happen when the frackman came within the proper radius of the barrel, so that it could set its state to visible. I then checked to make sure that if the frackman came within the proper radius, that the barrel would also disappear and be picked up by the frackman. Moreover, I tested to make sure that when a sonar was used, and the barrel was in a certain radius of the frackman, that the barrel's state would be set to visible.

class Gold

For the gold class, I tested it in a similar way as the barrel class. I tested what would happen if the frackman came within radiuses of 3 and 4 of the gold object, and that the proper courses of action would occur (i.e. being set to visible, and being picked up by the frackman, and being set to dead). In addition, I checked to make sure that when a sonar was used and the gold was within a radius of 12 that the gold would be set to visible.

class Bribe

For the bribe class, I tested what would happen when the user pressed the tab key (dropping a gold object). I made sure that the bribe was initialized in the proper location, and that the correct number of ticks would pass before the bribe was removed from the oil field. I then checked to make sure that if the bribe came within the proper radius of a protestor that it would be picked up by the protestor and its state would be set to dead to be removed from the oil field.

class Sonar

For the sonar class, I tested what would happen when the user pressed the z key. I made sure that if the frackman had a sonar object, that all goodies within a radius of 12 would be activated and set to visible on the oil field. I also checked to see what would happen if the frackman came within a radius of 3 of the sonar object, and should be picked up and set to dead on the oil field. I also checked to make sure that the proper time would pass before the sonar object would be removed from the oil field.

class WaterSquirt

For the squirt class, I tested what would happen when the user pressed the space bar. I made sure to test that the squirt would be initialized into the oil field properly, and that the squirt would start at the correct location. I also tested that the squirt object would travel the correct distance when it was fired (i.e. 4 squares in the given direction). I then checked what would happen when the frackman used the squirt object when it came into contact with dirt. I made sure that the squirt object behaved in the same way as the game provided (i.e. if 4 squares away from dirt, it should be visible in the game, but if 3 squares away from dirt, the sound should still play, but the squirt should not be visible). Similarly, I did the same test with the boulder and with the protestor. I also checked to make sure that the squirt would behave properly when it was near the edge of the oil field.

class WaterPool

For the pool class, I tested to make sure that the pool was properly initialized into the oil field (i.e. didn't spawn where dirt or boulder was present etc.). I also checked to make sure that the pool would disappear from the oil field after a given set of ticks had passed. Moreover, I made sure to check what would happen when the frackman came within a given radius of the pool, so that it should be picked up, and the proper number of squirts should be added to the frackman.

class StudentWorld

For this class, I tested that the game worked properly. I went around and picked up different objects (i.e. gold, barrels, pools etc.) to make sure that the score was properly being updated. I played through the given level, to make sure that the proper variables were being updated in the game (i.e. frackman's lives, gold, squirts, sonars etc). I then would make sure that every time the frackman either died or a given level was completed that the game would be cleaned up and initialized properly for the next level. In addition, I checked to see if objects were properly being initialized into the oil field at given ticks of the game, and that objects were properly being removed from the oil field when they were set to dead.