**PROJECT 4 REPORT**

**// FUNCTION DESCRIPTIONS**

**// ACTOR CLASS**

| **// pure virtual functions**  **virtual void doSomething()=0;** | I made these two functions pure virtual because the actor class itself is an abstract base class (an “Actor” object will never be instantiated-- only subclasses of Actor will be instantiated), and every derived class (every actor) has a doSomething method except for Earth that will be overwritten with an almost completely different function. Multiple of the actors have a beAnnoyed() class that will be overwritten with different functions. |
| --- | --- |
| **// accessor functions**  **bool isAlive()**  **{ return m\_isAlive; }**  **StudentWorld\* getStudentWorld()**  **{ return m\_studentworld; }** | * Bool **isAlive**() returns whether or not the actor is currently alive. Every actor can either be alive or dead, and other functions use isAlive() to determine whether or not to perform actions based on their living status. * **getStudentWorld**() allows derived classes to access the private member variable that is a pointer to the StudentWorld() class. This allows every actor to be tracked by StudentWorld, as well as access things like StudentWorld functions or the StudentWorld’s tunnelman pointer. |
| **// mutator functions**  **void setDead()**  **{ m\_isAlive = false; setVisible(false); }**  **Void beAnnoyed(int n)** | * **SetDead** allows any actor who calls it to set their isAlive to false, so other functions can access whether or not the actor is currently alive. It also sets visibility to false. I chose to not make this a virtual function, because every actor can either be alive or dead, and there aren’t any specialized functions for when they are setDead. * **beAnnoyed()** is NOT a virtual function despite how often it is repeated in the spec. The majority of classes do NOT get annoyed, so I just implemented beAnnoyed within the classes that get annoyed. For tunnelman, be annoyed just reduces m\_hit by the given amount. |
| **// helper functions**  **double calcDistance(double x1, double y1, double x2, double y2);** | **calcDistance** is a helper function in both the Actor and StudentWorld class so that every derived function can access it with convenience (without calling for a pointer to StudentWorld or vise-versa). This is used to calculate the radius between two points -- it allows for convenience when determining if an object should be generated in a given location, or if an actor should move to a given coordinate. |

**// EARTH CLASS**

| // constructor  Earth(StudentWorld \*sw, **int** x, **int** y)  : Actor(sw, TID\_EARTH, x, y, right, 0.25, 3) { } | The **Earth constructor** takes arguments for its StudentWorld and its starting X and Y coordinates so that the StudentWorld init() function can properly initiate a 2D vector of Earth objects with matching GraphObject m\_x and m\_y values. This grid works as the oil field and helps generate a map of pathways for the protesters to follow. |
| --- | --- |
| // virtual destructor  **virtual** ~Earth() { }    // accessor functions  **virtual** **void** doSomething() { }  **virtual** **void** beAnnoyed(**int** n) { } | Earth class has no implementation for any of its functions other than its constructor. This is because the Earth class, while derived from Actor to inherit isAlive() functionality, does not do anything (use doSomething()) or get annoyed (beAnnoyed()) during any ticks. Therefore, these functions are virtual but void of an implementation (not pure virtual, because Earth objects must be instantiated so Earth class can’t be an ABC). |

**// TUNNELMAN CLASS**

| **// constructor**  **TunnelMan (StudentWorld \*sw),**  **m\_gold(0),**  **m\_scharge(1),**  **m\_squirts(5),**  **m\_hit(10)**  **{ }** | My **TunnelMan constructor** takes a parameter to StudentWorld so that it can be added to/tracked by StudentWorld and accessed by all other classes/functions. It also initiates private member functions that track gold, sonar charge, squirts (water), and m\_hit count. These values can be accessed to update the GameStats screen, and are also accessed and updated when functions try to perform actions like pick up/drop gold, using a sonar charge, squirting the squirt gun, and losing hit points/health. |
| --- | --- |
| // virtual functions  virtual void doSomething();  virtual void beAnnoyed(int n) { m\_hit -= n; } | * **doSomething()** is a virtual function that is specialized for TunnelMan’s doSomething actions. If the TunnelMan isn’t dead (checked through Actor’s isAlive bool member), then the function will use a switch statement that takes input from the player’s keyboard to determine an action. For left/right/up/down keys, it checks conditions to either turn or move TunnelMan in the given direction. Other options like space, Z, and tab call other TunnelMan functions in order to use TunnelMan’s inventory items to perform actions. Finally, every time doSomething() is called, StudentWorld’s removeEarthHelper() function is called, which checks the new coordinates in case TunnelMan has moved, and sets the earth in these coordinates to invisible where applicable. If it does so, it will play a digging sound. * **beAnnoyed()** is a specialized virtual function, because TunnelMan behaves differently than the protesters when annoyed. TunnelMan can be annoyed by both the boulder and protesters, so it is more convenient to have a TunnelMan function that can be called by these classes then to have specialized annoyTunnelman() functions within both Boulder and Protester. beAnnoyed decrements m\_hit by the given amount. |
| // auxiliary functions  **void** collectGold() { ++m\_gold; }  **void** collectSonarKit() { ++m\_scharge; }  **void** collectSquirt() { m\_squirts+=5; }  **void** fireSquirt();  **void** illuminate(); | * **collectGold**() and **collectSquirt**(): adds gold/squirt to TunnelMan’s inventory when it encounters gold. This is in TunnelMan’s class instead of in Gold or Squirt class because only TunnelMan has access to the private member variable that tracks gold inventory. * **fireSquirt**(): I chose to put fireSquirt() inside TunnelMan rather than in Squirt class because only TunnelMan directly activates the squirt inside of its doSomething() specialized function. The Squirt class’s specialized doSomething function only handles travel/movement and being set dead. FireSquirt adds a squirt object to the oil field 4 units ahead of tunnelman’s current location using a switch statement and StudentWorld’s addActor() function. * **illuminate**() : I chose to put illuminate() inside TunnelMan rather than in Sonar Kit or Student World, because only TunnelMan calls illuminate directly inside its doSomething() specialized function. The Sonar Kit’s doSomething() specialized function only handles actions like being picked up or being set dead -- not actually activating/illuminating the field. It uses an iterator through all of StudentWorld’s actors to find those within a radius of 12 (with calculateRadius aux function), then sets these to visible. |
| // accessor functions  **int** getGold()  { **return** m\_gold; }  **int** getSonarCharge()  { **return** m\_scharge; }  **int** getSquirts()  { **return** m\_squirts; }  **int** getHealth()  { **return** m\_hit\*10; } | All of these accessor functions are used inside StudentWorld’s Game Stat updater. They’re necessary within TunnelMan because TunnelMan is the only class with access to its private member variables that are required for the game stat text. |

**// BOULDER CLASS**

| **// constructor**  **Boulder(StudentWorld \*sw, int startX, int startY)**  **: Actor(sw, TID\_BOULDER, startX, startY, down, 1.0, 1),**  **m\_stable(true),**  **m\_waiting(false),**  **m\_falling(false)**  **{ }** | The **Boulder Constructor** initializes private member variables that track whether or not the boulder is currently in a stable, waiting, or falling state. It initializes it to stable = true, which can then be changed during Boulder’s doSomething() state. |
| --- | --- |
| **// virtual functions**  **virtual void doSomething();**  **virtual void beAnnoyed(int n) { }** | * **doSomething()** checks what what state the boulder is in (stable, waiting, or falling), and will either check for earth beneath the boulder (stable), check if the appropriate number of ticks have elapsed to leave waiting state and enter falling state, * **beAnnoyed()** has an empty implementation because boulders can’t be annoyed |
| // auxiliary functions  **void** addTick() { ++m\_tick; }  **void** resetTick() { m\_tick = 0; }  **void** setStable() { m\_stable = **true**; m\_waiting = **false**; m\_falling = **false**;}  **void** setWaiting() { m\_stable = **false**; m\_waiting = **true**; m\_falling = **false**; }  **void** setFalling() { m\_stable = **false**; m\_waiting = **false**; m\_falling = **true**; }  **bool** =(); | * **addTick()**, **resetTick()** : These functions help update and reset the tick so that the boulder stays in the correct state for the appropriate number of ticks within doSomething. * **setStable, setWaiting, and setFalling** updates the status of the boulder and sets its other states to false (ex. setStable() will set isWaiting() and isFalling() to false. |
| // accessor functions  **bool** isStable() {**return** m\_stable;}  **bool** isWaiting() {**return** m\_waiting; }  **bool** isFalling() {**return** m\_falling; }  **int** getTick() { **return** m\_tick; } | * These accessors are all used by the doSomething() function to determine what state the boulder is in during the current tick (stable, waiting, or falling). The tick accessor helps to keep track of and update the tick appropriately, so that the boulder will remain in the correct state for the correct number of ticks |

**// PROTESTER CLASS**

| **Protester::Protester(StudentWorld \*sw, int imageID)** | The **Protester Constructor** initializes all tick trackers and the m\_hit count for protesters |
| --- | --- |
| **// virtual functions**  **virtual void doSomething();**  **virtual bool isHard()=0;**  **virtual void gotGold()=0;** | * **doSomething()** : This is marked virtual for the sake of good form because it is being inherited from Actor(), but the doSomething function is not being specialized by any classes derived from Protester. This is because the functions actually have almost everything in common, and can be differentiated using a bool variable (isHard) for features unique to the Hardcore Protester. * **isHard()** is a pure virtual function because the Protester class is an ABC that won’t be instantiated, and isHard must be specialized for both derived classes. This bool allows protester functions to identify between hard and regular protesters * **gotGold()** is a pure virtual class because the Protester class is an ABC that won’t be instantiated, and gotGold must be specialized for both derived classes (because hardcore protester pauses to admire gold, and the score increase is different) |

**// PROTESTER CLASS CONTINUED**

| **All of these accessor and tracker functions help other protester functions keep track of time for temporary states/variables/intermittent actions:**   * // accessor functions * bool isLeavingField() { return m\_leavingfield; } * int getTicksToWaitBetweenMoves() { return m\_ticksToWaitBetweenMoves; } * int getNumSquaresToMove() { return m\_numSquaresToMove; } * // auxiliary functions * // tick trackers * void decTicksToWaitBetweenMoves() { --m\_ticksToWaitBetweenMoves; } * void resetTicksToWaitBetweenMoves(); * void setZeroTicksToWaitBetweenMoves() { m\_ticksToWaitBetweenMoves=0;} * void decNumSquaresToMove() { --m\_numSquaresToMove; } * void resetNumSquaresToMove(); * void setZeroNumSquaresToMove() { m\_numSquaresToMove=0; } * void setLeavingField() { m\_leavingfield = true; m\_ticksToWaitBetweenMoves = 0; } * void beStunned(); |
| --- |
| **beAnnoyed**() is not a virtual function because the isHard() function allows the beAnnoyed() function to differentiate between a hard and a regular protester, and the majority of the beAnnoyed() function has the same actions. |
| **Actor::Direction getPerpTurn();**  uses a switch statement with the protester’s current direction in order to check for valid coordinates in the directions perpendicular from where protester is currently facing ; returns a valid perpendicular direction. This is implemented inside of protester for convenient/direct access to (x,y) coordinates and because the function is only used by protester. |
| **moveOne();** accepts a direction in which to check for valid coordinates if the protester were to move one forward using a switch statement. Both protester and squirt have separate moveOne function. |
| **bool isFacingTunnelMan();**  returns true if the protester is facing in the direction of tunnelman by testing and comparing the x and y coordinates of tunnelman and the protester |
| **Actor::Direction getClearPathDirection();** returns a direction in which the protester will have a clear path towards the tunnelman This uses a series of if statements that test if the orientation is horizontal, vertical, etc, and if there is earth or boulders between the two coordinates |
| **Actor::Direction generateRandomDirection(int x, int y);** uses a condition-less for loop that generates random numbers corresponding to directions until the coordinate is valid (there is no earth or boulder in the way of a movement). |

**// REGULAR PROTESTER CLASS**

| **RegularProtester(StudentWorld\* sw) : Protester(sw, TID\_PROTESTER) { setHit(5); }** | * constructor uses setHit() to set the hit count, because this can’t be initialized within the main protester class since it’s different for hardcore vs regular protesters |
| --- | --- |
| **// implementations of pure virtual functions**  **virtual bool isHard() { return false; }**  **virtual void gotGold();** | * isHard() is a virtual function because both derived classes are different-- regular protester is not hard, so isHard = false; * gotGold() is a virtual function because both derived classes have different reactions to being within a radius of gold. |

**// HARDCORE PROTESTER CLASS**

| **HardcoreProtester(StudentWorld\* sw) : Protester(sw, TID\_HARD\_CORE\_PROTESTER) { setHit(20); }** | * constructor uses setHit() to set the hit count, because this can’t be initialized within the main protester class since it’s different for hardcore vs regular protesters |
| --- | --- |
| **// implementations of pure virtual functions**  **virtual bool isHard() { return false; }**  **virtual void gotGold();** | * isHard() is a virtual function because both derived classes are different-- hardcore protester is hard, so isHard = true; * gotGold() is a virtual function because both derived classes have different reactions to being within a radius of gold. Hardcore protesters stop to admire the gold |

**// STUDENTWORLD CLASS**

**STUDENT WORLD CONTAINS ALL AUXILIARY FUNCTIONS OF OTHER CLASSES THAT ARE EITHER GENERAL TO MULTIPLE CLASSES, OR WHICH ARE IMPLEMENTED USING DATA STRUCTURES / VARIABLES PRIVATE OR DIFFICULT TO ACCESS THROUGH STUDENTWORLD**

| **CONSTRUCTOR**  **StudentWorld(std::string assetDir) : GameWorld(assetDir), m\_tunnelman(NULL), m\_earth(NULL), m\_actors(NULL), m\_nbarrels(0), m\_tick(0) { }**  The constructor initializes actors to null or 0 when possible. They will then be initialized properly through init() |
| --- |
| **VIRTUAL DESTRUCTOR**  **virtual ~StudentWorld() { cleanUp(); }**  **cleanUP();**  This destructor calls the cleanUp() function, which deletes tunnelman, deletes all remaining earth functions, and deletes all remaining actors. |
| **INIT()**  The init function initializes all tick trackers, initializes a new tunnelman and objects in the oil field. It sets the private data members of StudentWorld (m\_earth, m\_actosr, m\_tunnelman) equal to the new actors and the newly generated arrays full of new actors. |
| **FUNCTIONS THAT IMPLEMENT MOVE()**  **virtual int move();**  **void addNewActors();**  **void actorsDoSomething();**  **void removeDeadGameObjects();**  These functions are used directly by move to do the self-described actions. |
| **FUNCTIONS THAT ACCESS ACTORS**  **// actor accessor functions**  **TunnelMan\* getTunnelMan() { return m\_tunnelman; }**  **std::vector<Actor\*> getActors() { return m\_actors; }**  **Protester\* getClosestProtester(int x, int y); // returns a pointer to a protester within 3.0 radius or NULL**  **bool annoyProtestersWithinRadius(int annoy, int x, int y);**  **// used to access individual blocks of earth without accessor errors**  **Earth\* getEarth(int x, int y) {**  **if ((x>=0 && x<=63) && (y>=0 && y<=59))**  **return m\_earth[x][y];**  **else**  **return NULL;**  **}**  These functions all reside in StudentWorld because they acces StudentWorld’s private member variables, which include pointers to the game’s tunnelman and its full list of actors and earth objects. |
| **FUNCTIONS THAT MAINTAIN TICKS**  **void resetTick() { m\_tick = 0; }**  **void addTick() { ++m\_tick; }**  **int getTick() { return m\_tick; }**  **void decTicksBeforeAddProtester() { --m\_ticksBeforeAddProtester; }**  **void decTargetNumProtetsers() { --m\_targetNumProtetsers; }**  **int getTicksBeforeAddProtester() { return m\_ticksBeforeAddProtester; }**  **int getTargetNumProtetsers() { return m\_targetNumProtetsers; }**  **void resetTicksBeforeAddProtester();**  All of these functions are accessor and mutator functions that maintain ticks for generating new objects. These must be within StudentWorld(), because the move() function is what updates them. |
| **FUNCTIONS THAT SET THE GAME STATS DISPLAY**  **std::string formatGameText(std::string &s, int lvl, int lives, int hlth, int wtr,**  **int gld, int oil\_left, int sonar, int scr);**  **void setDisplayText();**  **collectBarrel()**  These functions are used to format and set the display text at the top of the screen, using inputs taken from tunnelman’s private inventory, or from GraphObject’s functions, or from private members of StudentWorld. This function is within StudentWorld because the data required of them is updated with every move and initialization, and the data is all accessible to StudentWorld. **collectBarrel**() is implemented in StudentWorld rather than in barrel or Actor because StudentWorld needs to keep track of the number of barrels in order to update Game Stats and determine the next level. TunnelMan has no use for adding oil to its inventory. |
| **compareBoulderHelper()** returns true if a falling boulder falls onto another boulder (if the bottom of the current boulder aligns with the top of any other boulder). |
| **FUNCTIONS THAT MUTATE THE MAP**  **bool removeEarthHelper(int x, int y);**  **void addActor(Actor\* actor) { m\_actors.push\_back(actor); }**  removeEarthHelper(int x, int y) → “removes” the earth in the 4x4 square around the given coordinates by using setDead(), which sets the earth invisible but does not delete it.  addActor() → safely accesses the m\_actors array to add new actors to StudentWorld. This is necessary within StudentWorld so everything in the game can keep track of every active actor. |
| **MAZE SOLVING FUNCTIONS:**  These 3 functions are used to (1) generate a “map” or “path” -- a 2D array filled with directions that are the optimal direction to move in to reach the provided x and y coordinates (destination) and (2) access the values stored in the generated map or path.   1. The map is generated through a maze-solving queue that first initializes a 2D array that is initialized to the size of the game window and filled with empty (“direction::none”) values. Then, a queue and a struct for coordinate variables is used to implement a breadth-first search through the map, checking for valid coordinates along the way (valid == no earth, no boulders). Every time a direction is viable and hasn’t already been traveled over, it is marked and filled with the optimal direction. This eventually fills the entire grid with directions. When the parameters are set in StudentWorld’s move function (either to the protester exit point (60,60) or to the current position of tunnelman), a map is generated. 2. The private member variables m\_protesterpathtoexit[][] and m\_protesterpathtotm[][] are both 2D arrays that are set equal to the “maps” generated by this function.   Therefore, later functions can use the values stored inside these two arrays in order to access the optimal direction to turn in when moving towards the exit or the tunnelman.  **// functions that help solve mazes and access directions generated through by maze solver**  **// creates a grid of directions towards goal position, given the goal**  **void generateProtesterPath(GraphObject::Direction map[][VIEW\_HEIGHT], int startX, int startY);**  **// accesses values stored in grid of directions (map) towards exit and towards tunnelman**  **GraphObject::Direction getDirectionExit(int x, int y) { return m\_protesterpathtoexit[x][y]; }**  **GraphObject::Direction getDirectionTM(int x, int y) { return m\_protesterpathtotm[x][y]; }** |
| **FUNCTIONS THAT CHECK THE ENVIRONMENT FOR OTHER OBJECTS**  **// functions that check if a boulder is under/on/in radius of given coordinates**  **bool compareBoulderHelper(int x, int y);**  **bool isBoulder(int x, int y); // returns true if the given coordinate contains a boulder**  **bool withinRadiusBoulder(int x, int y); // returns true if given (x,y) is within a 3.0 radius of any boulder**  **// functions that check if given coordinate is "valid", or there's no earth/boulders in the way**  **bool isValidMove(GraphObject::Direction dir , int x, int y);**  **bool isValidCoord(int x, int y); // returns if given coord overlaps earth/boulders**  **bool isEarth(int x, int y); // returns true if the 4x4 square with bottom-left (x,y) contains earth**  **// functions that calculate distance and radius**  **bool withinRadiusTunnelMan(double radius, int x, int y);**  **double calcDistance(double x1, double y1, double x2, double y2);**  **int getDistanceToTunnelMan(int x, int y); // returns the shortest distance from hardcore protester to tunnelman,**  All of these functions are implemented inside StudentWorld to have access to the member variables (lists of actors, earth, etc). They are all used to calculate whether or not the surrounding units contain boulders, earth, or are within radius to a protester/tunnelman/other actors. **calcDistance** uses recursion and the map for getting to tunnelman in order to calculate the shortest distance from protester to m\_tunnelman. |

**// SQUIRT CLASS**

| **Squirt (StudentWorld \*sw, int startX, int startY, Direction dir)**  **: Actor(sw, TID\_WATER\_SPURT, startX, startY, dir, 1.0, 1, false),**  **m\_dist(4) { setVisible(true); }**  The Squirt constructor takes arguments from the tunnelman when it is generating a squirt according to the “space” case. It passes in a direction and starting coordinate. |
| --- |
| **virtual void doSomething();**  doSomething() only handles the movement of a newly generated squirt object up to 4 units across the screen. It uses m\_dist, a private member variable that tracks how many units it has moved (through counting ticks). |
| **void moveOne();**  Checks if the squirt can move forward in whatever direction it is facing. This is implemented in the squirt class for the convenience of accessing direction and x and y coordinates. |

**// GOODIE CLASS**

| **// constructor**  **Goodie(StudentWorld \*sw, int imageID, int startX, int startY,**  **Direction startDirection, unsigned int depth, bool visibility,**  **bool pickupabiltyTM, bool pickupabiltyP, bool permanence)**  **: Actor(sw, imageID, startX, startY, startDirection, 1.0, depth),**  **m\_pickupabilityTM(pickupabiltyTM),**  **m\_pickupabilityP(pickupabiltyP),**  **m\_permanence(permanence),**  **m\_visibility(visibility)**  **{ setVisibility(visibility); }** | The parameters visibility, permanence, pickabilityTM, and pickabilityP, are used by all goodies at initialization and within functions in order to determine whether or not the goodies are visible, temporary, if it can be picked up by the tunnelman or picked up by a protester. |
| --- | --- |
| **// virtual functions**  **virtual void doSomething()=0;** | * **doSomething()** is a pure virtual function because a Goodie object will never be instantiated, so it should be an ABC |
| **// accessor functions**  **bool isVisible() { return m\_visibility; }**  **bool isPickupableTM() { return m\_pickupabilityTM;}**  **bool isPickupableP() { return m\_pickupabilityP;}**  **bool isPermanent() { return m\_permanence; }**    **// mutator functions**  **void setTemporary() { m\_permanence = false; }**  **void setPermanent() { m\_permanence = true; }**  **void setVisibility(bool tf) { setVisible(tf); m\_visibility = tf; }** | * All of these accessor and mutator functions are used to keep track of whether or not the goodie is visible, pickup-able by tunnelman, pickup-able by protester, and permanent/temporary. |

**// BARREL CLASS**

| **// constructor**  **Barrel(StudentWorld \*sw, int startX, int startY)**  **: Goodie(sw, TID\_BARREL, startX, startY, right, 2, false, true, false, true) { }** | The Barrel constructor doesn’t need to initialize any constants. |
| --- | --- |
| **// virtual void functions**  **virtual void doSomething();** | * **doSomething()** uses calcDistance to check if SonarKit is within radius to protester. If it is, then SonarKit will add water to the player’s inventory, increase the player’s score, and set itself dead. |

**// GOLDNUGGET CLASS**

| **GoldNugget(StudentWorld \*sw, int startX, int startY, bool visibility, bool pickupabiltyTM, bool pickupabiltyP, bool permanence)**  **: Goodie(sw, TID\_GOLD, startX, startY, right, 2, visibility, pickupabiltyTM, pickupabiltyP, permanence),**  **m\_tick(0)**  **{ }** | The **GoldNugget Constructor** takes arguments for starting coordinate, visibility, pickup-ability, and permanence. This allows init() to create invisible GoldNuggets that are pickup-able by tunnelman, and then allows tunnelman to drop visible goldnuggets that are pickup-able by protesters. |
| --- | --- |
| **// virtual void functions**  **virtual void doSomething();** | * **doSomething()** checks what state the goldnugget is in (visible? Pickup-able by tunnelman or by protester?) Then it uses calcDistance to determine if it is in the radius of a protester or a tunnelman, either adding gold to inventories or bribing the protesters. |

**// SONARKIT CLASS**

| **SonarKit(StudentWorld \*sw, int startX, int startY)**  **: Goodie(sw, TID\_SONAR, startX, startY, right, 2, true, true, false, false),**  **m\_tick(0)**  **{ }** | The SonarKit constructor sets permanence to false, making SonarKit temporary. M\_tick is initialized to 0 and then used to track time before the sonar kit self-destructs. |
| --- | --- |
| **// virtual functions**  **virtual void doSomething();** | * **doSomething()** uses calcDistance to check if SonarKit is within radius to protester. If it is, then SonarKit will add water to the player’s inventory, increase the player’s score, and set itself dead. It keeps an m\_tick tracker to know when to self-destruct. |

**// WATERPOOL CLASS**

| **// constructor**  **WaterPool(StudentWorld \*sw, int startX, int startY)**  **: Goodie(sw, TID\_WATER\_POOL, startX, startY, right, 2, true, true, false, false),**  **m\_tick(0)**  **{ }** | The **WaterPool** constructor receives its startX and startY parameters from a getValidWaterXY() function that generates a valid random coordinate without earth for the WaterPool to spawn in. The “permanence” parameter is set to false because it is temporary. The constructor sets m\_tick to 0 in order to start the countdown before the temporary WaterPool is setDead. |
| --- | --- |
| **// virtual functions**  **virtual void doSomething();** | * **doSomething()** uses calcDistance to check if WaterPool is within radius to protester. If it is, then WaterPool will add water to the player’s inventory, increase the player’s score, and set itself dead. It keeps an m\_tick tracker to know when to self-destruct. |

**// UNFINISHED FUNCTIONALITY AND BUGS**

**//** I am not aware of any unimplemented functionality or bugs.

**// ASSUMPTIONS AND AMBIGUITY**

// A list of other design decisions and assumptions I made

1. **Ambiguity of new Goodie dependence.**   
   On page 21 of the spec, the line “Assuming a new Goodie should be added, there is a 1/5 chance that you should add a new Sonar Kit, and a 4/5 chance you should add a Water Goodie” made me question whether or not adding a water goodie is dependent on adding a new sonar kit, and vise versa.   
   I made the assumption that these additions are independent of each other, or that, when addGoodie == true, there's a chance that you could add both a sonar kit and a water goodie during the same tick.

This means that, instead of writing an if/else chance like:

if(rand()%5==0)

add Sonar;

else

add Water;

I wrote two separate if statements:

if (rand()%5<1)

add Sonar;

if (rand()%5<4)

add Water;

1. **When to add the first protester.**On page 20 of the spec, the line “The first Protester must be added to the oil field during the very first tick of each level” made me question whether or not the first protester should be initialized within StudentWorld’s init() function or the move() function.  
   I decided to initialize it within an auxiliary function of move(), rather than init(), using an if-statement that checks if the tick count is currently 0.
2. **Probability of Hardcore Protester:**

Page 21 of the spec states that the probability of adding a hardcore protester = min(90, current\_level\_number \* 10 + 30). This made me question whether or not this is a percentage probability out of 100, or if it was a 1/probability chance.

I assumed that this means that this was a probability out of 100. This means that if the probability calculation returns 90, for example, that 90% of protesters will be hardcore protesters, and 10% will be regular protesters. To implement this chance, I used:

**int** addHardcoreChance = min(90, level\*10+30);

**bool** addHardcore = (rand()%100) < addHardcoreChance;

When addHardcore is true, it adds a hardcore protester // else it adds a regular protester. This means that the chance of adding a regular protester is dependent on the chance of adding a hardcore protester.

1. **Protester setDirection on its way towards exit**

The spec only specifies that a protester should advance towards the exit when it is in isLeaving() state-- it does not directly ask that we set the new direction of the protester so that their graphic image turns while they are advancing towards exit. However, the sample game does turn the graphic image with every move towards exit, so I added a line to my code to setDirection(getDirection()) for every tick the protester is moving towards (60,60). Alternatively, I could’ve added this line to my moveOne() function. I chose not to do so because the other function that calls moveOne() does not need to setDirection() (squirt always moves in one direction), so it would’ve caused unnecessary calls.

**// TEST CASES**

* **Boulder Class:** I used a cerr statement to test the coordinates generated by my helper function. I confirmed the boulders were never generated within the tunnel or too close to other actors. I visually tested if it was overlapping with earth objects. I confirmed that the tunnelman and other actors could not overlap with the boulder by moving the tunnelman towards the boulder from every direction. I tested the falling state by removing different amounts of earth from beneath the boulder. I used cerr to output the ticks to confirm that it was in waiting state for an appropriate number of ticks.
* **Earth Class:** I initialized the first level and several proceeding levels to confirm that the oil field of earth objects is always constructed fully, with the correct dimensions and a mine shaft down the middle. I used the debugger to read through the initialized vectors within my 2D vector to confirm that there would be no access errors later (no uninitialized coordinates).
* **TunnelMan Class**: I moved all around the map to check TunnelMan’s interactions with different objects. Using a cerr statement, I compared its coordinates to other objects on screen. I tested every key function (tab, esc, space, etc).
* **Squirt Class:** I used a cerr statement within each direction I could fire a squirt -- this statement (“squirted to the right/left/up/down”) confirmed that my switch statement was working as expected. Then I shot water at earth and boulders and cerred the coordinates to see when and where the water was setDead() (if the coordinates of the earth/boulders match the call to setDead()). I fired squirts until the gun was empty and after refilling the gun with water pools to test the interaction between the different classes. I used a cerr statement to compare the original and final coordinates of the squirt to confirm it was only travelling the appropriate distance.
* **Barrel of Oil Class:** I made calculations for several levels of how many oil barrels were to be generated. I compared these with the on-screen game stats update, the debugger m\_actor array, and my m\_oil private member variable. I also used cerr to check the validity of the coordinates generated by my random coordinate generator.
* **Gold Nugget Class:** I tested that onlytunnelman could pick up the originally generated gold-- and then I confirmed that only protesters could pick up tunnel man's dropped gold. I confirmed that tunnelman gained points if a protester was “bribed” using the gamestats update on screen. I confirmed that on-screen “Gld: “ game stat was consistent with what was being collected through game play. I also used cerr to check the validity of the coordinates generated by my random coordinate generator.
* **Sonar Kit Class:** I used cerr to output the current game tick, and compared the tick with the expected frequency of sonar kit generation. I used the sonar kit to illuminate nearby objects, using cerr to get the coordinates of objects to confirm that everything in a 12.00-unit radius was set to visible.
* **Water Pool Class:** I used cerr to output the current game tick, and compared the tick with the expected frequency of sonar kit generation. I collected water and confirmed that the game stat update was consistent with what I was collecting. Then I used the water to fire the squirt gun, to test the interaction between different classes.
* **Regular Protesters Class:** I watched the protesters to confirm that they didn’t have irregular movements (too fast, too slow, illogical turns). I used the squirt gun and boulders (along with a cerr statement of the m\_hit score) to test whether m\_hit scores would decrease as expected, and if the stunned state and isLeaving() states behaved as visually expected. For example:

cerr << "original m\_hit: " << m\_hit << endl;

m\_hit -= amount;

cerr << "after getting hit: " << m\_hit << endl;

I used a cerr statement to test the ticks of these functions, to see how long the protesters remained in a stunned state and in a normal resting state, and how often each type of protester was being generated. I also re-confirmed that the protesters were being added to the m\_actors array by using the debugger to look through the StudentWorld m\_actors array.

I also used a cerr statement to test each possible case of movement, so I could compare if they were making actions at the correct rate and performing the correct actions. For example, I used “cerr<<"i'm resting"<<endl; “ inside my loop checking for m\_ticksToWaitBetweenTurns, and “ cout << "is turning in random direction: " << getDirection() << endl; “ to make sure that it was turning in logical directions.

To test if the protester would turn towards the tunnelman in a straight line of sight, I added cerr statements to each directional possibility ( EX: cerr << "protester ABOVE tunnelman, turn DOWN" << endl; ) that would let me know if the correct pathway was activating when I moved tunnelman directly under the protester. I confirmed that the protester wouldn’t turn unless the path was clear.

* **Hardcore Protesters Class:** I did everything I did with the regular protesters class, in addition to using my TunnelMan to walk around the HardCore protester to see if it could sense and then follow the TunnelMan. I used cerr to output coordinates to confirm that hardcore protester would only track TunnelMan once within a calculated radius.