REPORT: Christina De Cesaris

1. 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

Report

1. StudentWorld Class: most of this project was bridging the communication between the actors and the world. Many of the member functions in studentworld served as means to give an actor a item, return the pointer of an actor, and determine possible routes on the map to either betaken or if possible to take.
2. int randIntGeneratorX(); //for inserting at random locations, generates a value that is appropriate for a x/row coord on the grid
3. int randIntGeneratorY();//for inserting at random locations, generates a value that is appropriate for a y/col coord on the grid
4. bool personCanMove(int x, int y, bool canMove) const; //if boulder or earth this is no, bool checks via using atEarth and the isboulder bool from the boulder class
5. bool atEarth(int x, int y) const; //determin e if the current cords are on earth object, checks if the earth matrix pointer at this coord is null or not
6. bool removeEarth(int x, int y); //deletes pieces of earth and sets ptrs to null, used for tunnelman moving through the earth and boulder placement
7. void addBarrel(int amt);//increases world’s member barrel numbers
8. TunnelMan\* findTunnelMan(Actor\* act, int rad) const; // one of the most vital functions in the design, findtunnelMan returns a pointer pointing at our tunnelman if he is in a particular radius. If he is found in said radius, than other objects can behave accordingly
9. Protestor\* findProtestor(Actor\* act, int rad) const;//copy of findTunnelman but returns ptr of protestor for picking things up, similar to findtunnelMan but protester implementation
10. TunnelMan\* getTunnelman() { return m\_tunnelman; }//returns our tunnelman pointer, useful for accessing players current coordinates if needed for our actors
11. void giveSonar(int num);//gives tunnelman a sonar in his inventory when called, allows sonar class to use getworld and indirectly increase inventory
12. void tunnelManUseSonar(TunnelMan\* man);//allows tunnel man to use sonar, this function utilizes withinRad to make visible all the actors that are invisible within the current radius, it runs through to see if the coordinates of those actors are within the radius
13. void giveGoldTunnelman(int num, TunnelMan\* a); //increases the tunnelmans inventory of gold
14. void addActor(Actor\* a);//gives an actor back to the world, this is used for when an item in the actor class needs to be spawned, for example, when the tunnelman drops gold it is given back to world to be spawned
15. bool withinRad(int x1, int y1, int x2, int y2, int rad) const;// calculates with the given coordinates of two are within a certain radius, used for determining relative positions of actors
16. bool tunnelManInSight(int x, int y, Actor::Direction dir) const;//recursive function which determines if the given position and direction of a protestor is facing the tunnelman, it calls itself forward in the direction the protestor is facing and if we see the tunnelman present in those spaces, then it returns true
17. //void dropGoldNugget();
18. void annoyAllNearbyPeople(Actor\* person, int rad, int points, bool annoyPlayertoo );//designed to annoy all nearby protesters and tunnelman, in the case of boulders, they can annoy both player and the protestors, this function uses within rad as well and determines the amount of damaged points to be taken from the actor.
19. bool noOverlap(int actX, int actY);//checks to see if a to be spawned obj is at least 6 away in radius, as well uses withinRad and purposed for placement of items
20. Actor::Direction getProtestorDirection(Actor\* p) const;//vital function for returning the direction a protestor should move in inorder to leave the oilfield, it uses a private helperfunction called canLeaveOilField which determines if a position is feesible to move in for protestor
21. void giveWaterTunnelman(int num);//updates tunnelmans water inventory
22. bool canSpawnPool(int x, int y) const; //can only spawn pool in 4 by 4 square without earth or another object and in a tunnel, determines if a pool can spawn

Actor:

class Actor : public GraphObject {

public:

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

virtual ~Actor();

virtual void doSomething()= 0; //made into a ABC to insure that all classes will have their own virtual doSomething defined. If they do not, they must rely on the default of their parent such as my unfinished hardcoreprotestor ☹

virtual bool blocksMove() const { return false; } //to be over written by objects that DO block movment, aka boulders, is present in actors for when I need to iterate through all the current actors to determine if the one in question can move in a particular area

bool isAlive() const; //returns the life state of actor in question

void setLifeState(bool liveState);//sets the life state

StudentWorld\* getWorld() const; //returns a pointer to the student world allowing actor member functions to have a way of influencing the changes in the student world, such as an increasing the player’s score when a boulder hits someone etc etc

void newPos(int& x, int& y, int distance, Direction dir) const;//function was intended for shorthanding the new coordinates of an actors next move, it came in handy while moving the protestors because it returns what the new position would be in a certain direction which then can be used in moveTo, or direction checking functions to determine if a certain position is valid.

virtual bool isBoulder() { return false; }//the following bool functions are identification functions for each actor, when I iterate through the actors vector, there functions came in handy if I needed said actor to do something specific in the world

virtual bool isPerson() { return false; }

virtual bool isProtestor() { return false; }

lass Earth : public Actor {

public:

Earth(int startX, int startY, StudentWorld\* world);//set it visible

~Earth();

virtual void doSomething(); //earth really doesn’t do anything

class Person : public Actor {

public:

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

virtual ~Person();

int getHealth() const; //return health of person

void setHealth(int health);//sets health of person

virtual void annoyPerson(int amt) =0; //abc class because it is intended to be overwritte by the protestors and tunnelman who are both people, annoying them has different results

virtual bool isPerson() { return true; }//overwrites the original to true, this was inline

class TunnelMan : public Person{

public:

TunnelMan(StudentWorld\* world);//init tunnelman and his vars

~TunnelMan();

virtual void doSomething();//huge function, if health is zero dooes nothing other wise, determines if the tunnelman is moving toward earth and appropriately calls functions to remove the earth he is moving into according to his size (playing sounds respectively), checks for user input keys and responds to each key, if KEYUP is pressed than the function calls TM’s private userMove fuction which sets his direction and moves him in said direction one unit, same for interaction with other objects, or dropping gold, it will give a gold pice to student world etc

int getSonars(); //returns TMs sonar count

void setSonar(int num);//sets TMs sonar count

int getGold();//returns gold amt

void setGold(int num);//sets gold amt

int getWater() { return m\_water; }//returns water amt, at this point I decided to go toward inlining trivial functions to save space and chaos in my .cpp

void setWater(int num) { m\_water += num; }

virtual void annoyPerson(int amt); //tunnelman specific annoyPerson function, important for subtracting his health, I tried to play the player annoyed sound but I don’t believe there really is one (it isn’t in the specs but I thought it would sound cool oh wells haha)

lass Protestor : public Person {

public:

Protestor(StudentWorld\* world, int id);//inti protester and vars

virtual ~Protestor();

virtual void doSomething();//checks if alive if not than does nothing, if health is less than zero, set to leaving, runs through the current ticks and returns until current ticks is zero. If the protestor is in the leavingthefield state, set his lifestate to false if he is in the proper exiting corner of the map, otherwise, use getProtestorDirection to return the direction he must take to make his way out of the oilfield, set the dirction to the returned direction and then move one unit in said direction, count down our shoutingwaitingticks and our perpendicularturnticks, if we are in the 4 units of TM and the shoutingwaitiingticks is zero and he is in the correct direction(facing him) shout to annoy him and reset the waiting ticks. Otherwise,check if TM is insight of the protestor given the protestors current facing direction. If we find tunnel man is indeed in sight, move towards him (earth should obscure view). If TM is not in line of sight, choose a random direction and check if that position is valid via personcanMove(), if we find a valid direction and our perpendicularturnticks is zero, then we attempt to turn(this also depends on what direction so up or down, right or left, will choose a perp direction randomly but also only chooses if such direction is possible. If TM is out of sight and perpendicularturn ticks is not zero, then move in the original chosen and possible random direction. If we are just spawning however, I make the protestor move left initially so he can walk onto the field.

bool isleavingField() const { return m\_isleaving; }

void setLeavingStats() { m\_isleaving = true; }

void setSquarestoMove(int squ) { m\_squarestomove = squ; }

int getSquarestoMove() { return m\_squarestomove; }

int getCurrentTicks() const { return m\_currentticks; }

void setCurrentTicks(int tic) { m\_currentticks = tic; }

int getWaitingTicks() const { return m\_waitingticks; }

virtual void annoyPerson(int amt);

virtual bool isHardCore() { return false; } //rgular is false, useful in determining points

//virtual bool blocksMove()const { return true; }//not needed

virtual void bribe(); //for bribing with gold, plays the sound and gives score to the tunnelman,

virtual bool isProtestor() { return true; }//overwriten protetor class

class StillActor : public Actor {

public:

StillActor(int imageID, int startX, int startY, Direction startDirection, float size, unsigned int depth, StudentWorld\* world);//for all the classes that do not move, these are the pickupables

~StillActor();

bool isVisible() const;

void setTicks(int ticks); //all of these have a tick counter to init

int ticksRemaining() const; //retrieves ticks of object

void makeStillActorVisible(); //turns still actor visible within its own class, this was put in place to mediate a bug sadly, it allows for object to come visible when TM is close by

OilBarrel(int startX, int startY, StudentWorld\* world);//also adds a barrel to the studentworld

~OilBarrel();

virtual void doSomething();//if tunnel man is nearby, become visible, if he is in pickup distance via using findTunnelman within a certain rad, then become invisible and call the studentworld to increases tunnelman’s inventory

class Sonar: public StillActor {

public:

Sonar(int startX, int startY, StudentWorld\* world);

~Sonar();

virtual void doSomething();//similar to oilbarrel in terms of allowing tunnelman to pick it up, it also uses getWorld() to give TM a sonar, it disappears after it has been obtained as with oilbarrel. However, sonar is created visible and is only present for a certain amount of ticks which it decrements with each call

};

class WaterPool : public StillActor

{

public:

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

~WaterPool();

virtual void doSomething();//also very similar to oilbarrel and even more so, it will disappear if the TM is within appropriate distance and then call the world to give TM water;

Goldnugget(bool spawned, bool visability, bool perminate, bool forTunnelman, int startX, int startY, StudentWorld\* world);//the key to goldnugget was using many Boolean parms to determine who could pick it up, if it is to be visible and if it is going to spawn or not

~Goldnugget();

virtual void doSomething();//its functionality is almost identical to that of sonar in terms of exisiting for a certain amount of time after being dropped. The parms allow it to be picked up by either TM or protestors. If it is picked up by protestors it can use getWorld() to call the bribe function. If it is originally present, and not dropped it will not disappear.

class Boulder : public Actor {

public:

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

~Boulder();

virtual void doSomething();//checks if dead and if so return, otherwise, if the spaces below it (in the 4 units) are not earth, then the boulder sets itself to a waiting state which decrements time, once that time is spent, it moves into a falling state, plays the soundeffect, and is set to move downward as long there is no earth directly below it, if there is earth in any of the 4 areas below it, then it is set to dead and disappears.

virtual bool blocksMove() const { return true; }; // cannot pass through bolders

virtual bool isBoulder() { return true; }

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

~Squirt();

virtual void doSomething();//if dead then return, otherwise looks for a protestor, if it finds one then it is set to dead after it annoys the protestor. Otherwise if its distance is zero it also is set to dead. If not dead, then it checks if it can move in the direction TM is facing, if it can move there (no earth or boulders etc), then newPos is called to update its current position and it is set to move until distance becomes zero or it runs into something, at that point, it disappears.

lass HardCoreProtestor : public Protestor {//UNFINISHED

public:

HardCoreProtestor(StudentWorld\* world);

~HardCoreProtestor();

virtual bool isHardCore() { return true; } //rgular is false

//virtual void bribe(); //not needed via additon of isHardCore

//virtual void doSomething();//ran out of time

int getStaringticks() { return m\_staringticks; }

int setStaringticks(int num) { return(m\_staringticks + num); }

Unfortunately, I was unable to implement the searching behavior of the HardCoreProtestor due to time constraints, it is treated as a regular protestors except it has more health, gives more points when leaving or gets gold, and does not leave if it picks up gold.

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 54 Hardcore Protester doesn’t work correctly yet so I just treat it like a Regular Protester right now.”

My HardCoreProtestor does not have a proper searching alg for honing in on the tunnelman, it should not leave when bribed and the points given to the tunnel man are more when it is bribed

Squrits: my squirts work but do not give points for when the tunnel man makes a protestor leave, the protetors do leave and if I had time to write a function for it, I would update my annoy person function and have a bool for is squirt, when I squirt a protestor it is true, then, when I run through the annoyperson, it can determine if that protestor is dead or not and hardcore or not and then give the appropriate points.

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

The most difficult design was the protestor’s random movement and leaving the oil field. I decided to do a recursive check for solving the maze which required I add private helperfunctions to my student world class. I created a matrix of integers and marked the spots previously visited, and help search for a direction of possible exit. I had a difficult time figuring out how to know if the tunnelman was within distance but the findTunnelMan\* function became very useful in aiding. The implementation of it could be improved. I basically return a ptr at the tunnelman and if that is null than he is not in the vicinity. If it is not null, then the said item can disappear and behave accordingly.

I found it was best to design the project with questions and mapping out the planning:e.g.

What does my actor need or need to do that must be mediated through studentworld (such as location of the TM or other actors, or if we are atEarth)? Then creating functions to accommodate. How can I use studentworld to allow for “communication” between my actors. How can I get my own actors to behave in a particular manner etc. I often found myself having to add in functions as I went along, realizing that there was something I needed that I didn’t initially plan. In most cases these functions were trivial and came up as I realized I needed to include other specs..

4. Testing

TunnelMan: TunnelMan’s testing consisited mainly of intentionaly creating objects within student world (often in easy to access locations) and having TM interact with the objects such as moving forward to pick them up. I ended up making my sonar reveal all items for testing purposes as well. This way I could go straight to the barrels after I was certain they would appear otherwise. I was able to save time. Tunnelman’s movements were tested with my keyboard accordingly. I would often spawn an item or protestor and then see how they interact with my character. If there was crashing issue, I would use debugger and check any loops that might be the culprit.

Earth: Earth could be tested after tunnelman could move. I used him to determine if earth would be removed if he walked into it etc.

Sonar: To test sonar, I initially spawned in my init function so I could test if tunnelman could pick it up and use it via ‘z’. I tested it after I was sure that other objects would spawn if TM moved close to them. I would spawn in inti, barrels and then use sonar to confirm that these items appear.

Boulder: I used inti to spawn a boulder in the top of the earth, I then used tunnel man to remove the earth and observed the boulder’s actions. I had complications with getting the sound to play accordingly and ended up scraping the doSomething for another one. If I spawned the boulder in a shaft I expected it to fall. If I spawned a protestor beneath a boulder in the shaft, I expected it to bonk him and same with TM.

Goldnugget, waterpool, and oilbarrel: All were tested in a similar fashion and for sake of time, I spawned them in the init function for testing. I placed them in locations I knew they would appear in to see if they appeared and were picked up. I relied a lot on visual clues for testing. I also spawned to test if the amount of time an item was present ran our or not. I used the displayText() to tell me what items I had picked up. It was very useful because the text could let me know if I actually did add an item to the tunnelman’s inventory or not and took it away when used.

Squrits: Tested after the implementation of protestors and water. Tested by spawning protestors and shooting at them to be honest. I also tested it in nonblocked directions and blocked directions so I would know if it disappeared when it hit somewhere it couldn’t move.

Protestors: The most difficult class to test. I spawned a protestor in init and used TM to test their behavior. For example, if I coded in the shouting implementation, I would use TM to approach the protestors and confirm it worked. The same was used for the inlineofsight movement. For leaving the field, I would lure them into a location then drop a gold piece and see if they could find their way out. After I got squrits to work, I just squirted them until they ran away correctly.