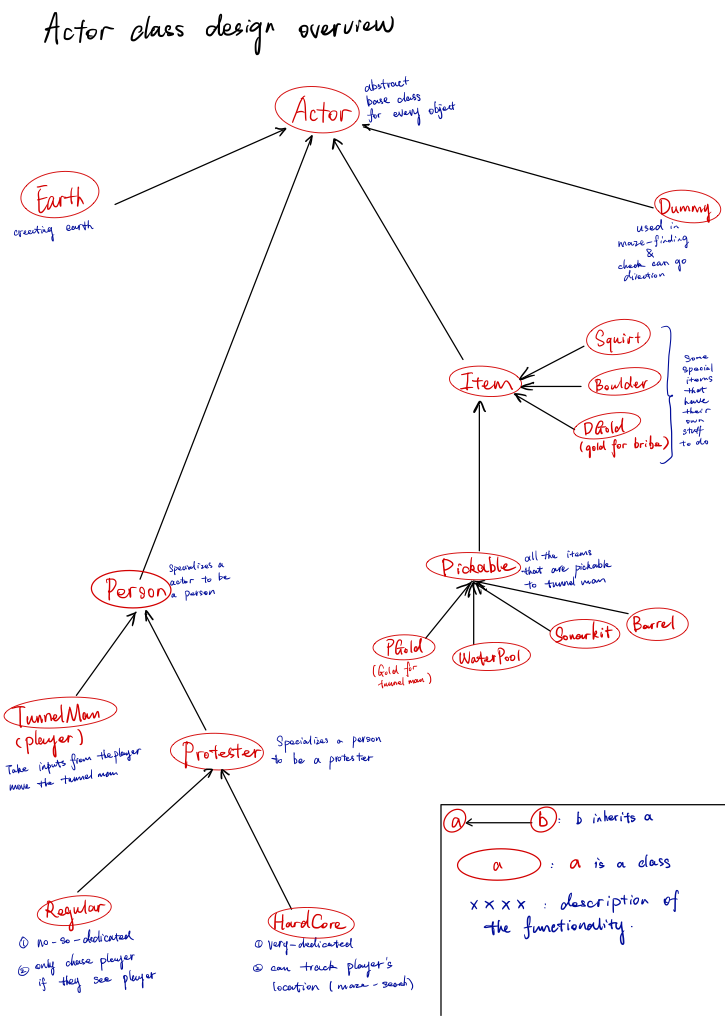
Project 4 Report: TunnelMan

Litian Liang

1. **High level description about class and member function design.**



class StudentWorld:

This class is responsible for running the game.

class StudentWorld : public GameWorld{

public:

inline int max(int a, int b){

return (a > b) ? a : b;

}

Reason: this function is useful when getting a bigger number from two numbers, increases readability.

inline int min(int a, int b){

return (a < b) ? a : b;

}

Reason: same as above

StudentWorld(std::string assetDir);

Reason: this constructor will initialize

1. a vector of Actor pointers that contains every Actor in the game (including earth)

2. a 2d array that contains the Earth pointers

3. a counter that counts the number of oil left in the oil field

4. a counter that counts the last time that a sonar kit is added into the game

5. a counter that counts the last time that a protester is added into the game

6. a counter that counts the number of protesters that are in the game

~StudentWorld();

This method will deallocate the dynamic array for Earth pointers

virtual int init();

This method will (in this specific order):

1. initialze earth

2. initialize boulders

3. initialize oils

4. initialize golds

5. initialize tunnelman

virtual int move();

This method will (in this specific order):

1. display header text

2. check if player is dead

3. add runtime goodie using runTimeGoodie()

4. add protester using runTimeProtester()

5. let every Actor in the vector of Actor pointers do something

6. check how many oil left in the oil field: if 0, level finished

7. continue game

virtual void cleanUp();

This method will deallocate all the objects in the game (including earth)

int& oilLeft();

This method will return the oil count

// get the tunnelman ptr

TunnelMan\* getPlayer();

This method enables the actors to know where is the player right now to react accordingly in their doSomething() (ex: protester shout to tunnelman or gold nugget picked up by tunnelman)

// get everything in the game

std::vector<Actor\*>& getevery();

This method enables the actors to know the status and location of all other actors and react accordingly. (ex: protester bribed by picking up a gold nugget dropped by tunnel man)

// get the earth field

Earth\*\*\* getfield();

This method enables the actors to directly get that if a specific position is occupied by earth. (ex: Dummy cannot go into earth, TunnelMan can dig through earth (setting this earth to dead using setDead()))

private:

void initRandomLocation(int& x, int& y, int xfrom, int xto, int yfrom, int yto);

This method will take in a location range specified by xfrom, xto, yfrom, and yto, and then generate a random coordinate x, y base on the range in the argument. This method is used to generate gold and barrel before the game started. (This method will not generate an illegal coordinate because illegal situation is handled in this method)

void runTimeGoodie();

This method will only generate Sonarkit at top left corner, and only generate water pool on streets dig up by tunnel man (not overlap with earth).

void runTimeProtester();

This method will generate protesters if found needed by checking the total number of protesters that are there in the game.

std::vector<Actor\*> everything; All the actors in the game

Earth\*\*\* field; A 2d array of earth objects

TunnelMan\* p; // tunnel man ptr

int oilCtr; Total oil left in game

int lastTimeSonar; last time a sonar is added to the game

int lastTimeProtester; last time a protester is added to the game

int totalProtester; total number of protesters in the game

};

Actor: (derived from GraphObjects)

This class is an abstract class in order to use the polymorphism in the vector of actor pointers. Every time a new Actor pointer is pushed back to the vector of everything in StudentWorld class, it will be auto-casted to this base class Actor.

class Actor: public GraphObject{

public:

// constructor

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

// destructor

virtual ~Actor();

// check current state

bool isAlive() const;

void setDead();

// take in a direction, get the deltax and deltay needed

void getNextStep(GraphObject::Direction dir, int& dx, int& dy) const;

Reason:

In the game, there are many operations involving moving objects around in different directions. This method takes in direction, and tells the object that how it need to decrement or increment its x and y coordinate.

// get absolute distance between another Actor

double getDistance(const Actor\* other) const;

Reason:

In the game, there are many operations checking for distance between this object and another object.

// get a opposite direction

GraphObject::Direction getOpposite(GraphObject::Direction dir);

bool facingPlayer();

Reason:

In protester class, this member is used in the maze finding algorithm. You can see detailed reason why I need this method there.

// if this actor overlaps another actor

bool overlaps(Actor\* other) const;

Reason:

Some actors cannot go on earth or boulder. This method simply checks if these two actors overlaps with each other.

// annoy

virtual void annoy(int howmuch);

Reason:

To use polymorphism, all actors can be annoyed. But not all actors have action if they are annoyed. ex: Earth's annoy is just empty, doesn't do anything. This is not a pure virtual function because most objects (ex: Earth) can not be annoyed. Only protesters and tunnelman do. So it is empty in actor, but it is implemented in protesters.

// become rich

virtual void becomeRich();

Reason: exact same as annoy. This is not a pure virtual function because most objects (ex: Earth) does not becomeRich. Only protesters do. So it is empty in actor, but it is implemented in protesters.

// can this Actor dig through earth?

virtual bool canDig() const = 0;

Reason:

Some actors are able to go through earth, some are not. This method is used in canGo() method to decide that if an Actor can go through something. This is pure virtual because it's going to return different values in different classes.

// distinguish items and actors

virtual bool isActor() const = 0;

Reason:

If something is only happening to a person, not an item. This method will return true. The name is confused because once I used Entity and Actor to name Actor and Person classes. This is a pure virtual function because it's going to return true and false in Person and Item. False and false for Dummy and Earth.

// cango

bool canGo(GraphObject::Direction dir);

Reason:

There are many circumstances that some object need to check if it can go in a specific direction.

// everything doSomething every tick

virtual void doSomething() = 0;

Reason:

Use polymorphism to let every object do its routine.

// get size

float entitySize() const;

Reason:

Check the size of this actor, used in canGo() and overlaps().

// getworld

StudentWorld\* getWorld();

Reason:

In many circumstances like annoying the surrounding people, they need to know where all other objects are. So they need to use this pointer to get the vector that contains everything. Sometimes an object need to see what level it is right now, just use this pointer to get the information from the StudentWorld class.

private:

// worldptr

StudentWorld\* worldptr;

A pointer that points to student world.

// current state

bool m\_alive;

This object is alive or dead.

// size

float m\_size;

The size of this object.

};

class Earth: (derived from Actors)

This is the class for creating Earths. It doesn't do much. The doSomething and annoy function is empty.

class Dummy: (derived from Actors)

This class is useful in many circumstances.

1. canGo() in Actor class decides if an object can go in a direction using dummy to see if a Dummy go this way will overlaps with Earth or Boulder.

2. canSeePlayer() in Actor class just creates many dummies in four different directions to see if any dummies overlaps with player.

3. dirTo() in Protester class is a queue-based maze finding function that uses dummy to try to go from the destination to different directions to see if an dummy can finally have the same coordinate with this object.

class Item: (derived from Actors)

This abstract class cut down some intermediate functions like will return false in isActor(). In other words its implementation is specialized 1 step towards the items like squirts and gold, but not fully specialized to be an actual object.

class Pickable: (derived from Items)

This class is specialized for the items that are pickable to tunnelman (not protesters). The derived classes of pickable will call the pickable version of doSomething() before they call on their own version of doSomething().

ex:

void Pickupable::doSomething(){

if (getDistance(getWorld()->getPlayer()) <= 3)

setDead();

}

void Barrel::doSomething(){

// check dead

Pickupable::doSomething();

if (!isAlive()){

getWorld()->oilLeft()--;

getWorld()->playSound(SOUND\_FOUND\_OIL);

getWorld()->increaseScore(1000);

}

// check visibility

if (getDistance(getWorld()->getPlayer()) <= 4){

setVisible(true);

}

}

class PGold, Sonar, WaterPool, and Barrel (derived from pickable)

In the constructor, PGold (pickable gold to tunnelman) and Barrel are initialzed to not visible. WaterPool and Sonar initialized to visible. They play different sounds when they die. There is not much things they do except from checking if they died, playsound and change the number of oil left (Barrel) or increment the water count (WaterPool) of the tunnelman.

class Boulder, Squirt, and DGold:

These objects have their special implementations and they cannot be picked up by the tunnelman. So they are directly derived from Item class.

class Boulder (derived from item):

Boulder can't dig. In its doSomething() method, it will check to see if it canGo(down), which is defined in base Actor class. If it is not falling, it will become to fall after few seconds if it canGo(down). If it is falling, it will stop falling and setDead() if canGo(down) returns false.

class Squirt (derived from item):

Squirt object will first check if it can be created in the constructor. If the squirt does not have enough space (4 units) (this is done by creating a Dummy object in front of the tunnelman to see if the Dummy object is overlapping with Earth or goes out of bound), it will immediatly setDead(). So that it won't do anything in the doSomething() method.

In the doSomething() method, it will check if it canGo(getDirection()). If so it will moveTo() the next position. If not it will setDead(). Or if it finished travel 4 units on its direction, it will setDead().

class DGold (derived from item):

This is a special object that can only be picked up by protesters, there is no other object in this game can be picked up by protesters. It will check if there are protesters around it. If there is protester around, it will call its becomeRich() method to let it become rich and immediatly setDead(). Otherwise it will decrement lifetime.

It has a private variable (lifetime) that can track the ticks that it is created. It will setDead() if the lifetime become 0.

class Person (derived from actor):

This abstract class goes 1 step towards the implementation of a person (protesters and player). They all have hp (hit points), and they all return true in isActor() function to let other objects know that they are person, not item, so they can respond accordingly.

class Person: public Actor{

public:

Person(StudentWorld\* world, int imageID, int startX, int startY,

GraphObject::Direction startDirection, int hp);

bool isActor() const;

bool canDig() const = 0;

Reason:

Some people (tunnelman) can dig, some (protesters) cannot. This pure virtual function will return true in TunnelMan class and will return false in Protester (and its derived) class.

// annoy and get hp

virtual void annoy(int howmuch);

Reason:

All people responed to annoy by decreasing their hp, but they may have other specific implementation (leave the field or be stuned). These implementations can be done after they call the superclass (Person) version of annoy before they call their own.

int getHP() const;

Reason:

All people have hp, they need to know sometimes that their health.

// dosomething

virtual void doSomething() = 0;

Reason:

There are not much similarity between player and protester, this doSomething() does not need to be implemented, so it is pure virtual.

virtual ~Person();

private:

int hp;

};

class Protesters (derived form Person):

This class will implement many important specializations for the Protesters.

class Protester: public Person{

public:

Protester(StudentWorld\* world, int imageID, int startX, int startY, int hp);

bool canDig() const;

virtual void doSomething() = 0;

void annoy(int howmuch);

Reason:

Whenever protester got annoyed, their hp got deducted. There is no exception to this in any situation. But depending on how much it annoys this protester (lethal or not lethal (deduct hp below 0) ), it will decides if this protester need to be stuned with calling stun(). Because polymorphism, the Regular / HardCore version of annoy will be called first, but both of them will call the superclass (Protester) version of annoy before they call on their own version.

virtual void becomeRich() = 0;

Reason:

Regular Protester and HardCore Protester have different behavior when they "become rich" (bribed). Therefore, this is a pure virtual function.

virtual ~Protester();

protected:

Reason:

These methods will only be called by the protester themselves, not from the outside world. However, they will be called in the derived classes (Regular and HardCore). Thus, they are implemented as protected methods.

// stun the protester

void stun();

Reason:

Only a non-lethal annoy (by the squirt gun) can stun the protester. This method is called in the annoy method of Protester only if this annoy didn't deduct hp below 0.

// get the direction if a protester wants to go to this x,y point

GraphObject::Direction dirTo(int x, int y, int& moves);

Reason:

This is a queue-based maze-finding algorithm. In many circumstances (find the way to exit when the protester is exiting or a hard-core protester is tracking the player's position), the protester need to know which direction to go in its current tick.

How to use this function:

This function takes in a destination coordinate (x, y), and returns the direction that this protester should go this tick. You can also pass-by-reference a int argument to get the total steps that you need to go to your destination (x,y).

Detail implementation:

This algorithm first creates a dummy at the destination coordinate and push it into the queue. Then it evaluates all the directions that it can go and creates dummy objects in the direction where it can go and push them into the queue. Third step is to check if this dummy object has the exact same coordinate (x,y) as the protester, if true, return an opposite direction object of the direction that this dummy is facing right now (this is the direction that the protester should go). After evaluating each dummy object in the queue, it will be popped out from the queue and go back to the first step.

Count total step is done by using another queue, when a dummy is pushed in the queue, the function will get the step count from this queue, increment it by 1, and push it back into this queue again.

// check on which direction can protester see player (none if can't see)

GraphObject::Direction canSeePlayer();

Reason:

The protester need to check if it can see the tunnel man every tick. This function will return the direction if the protester can see player, return none if cannot. This is done by creating 4 dummy objects and let them walk towards their own directions to see if they can overlaps with the tunnel man by using overlaps() or they hit the bound or Earth they will stop.

// get the leaving status of this protester

bool& leavingStatus();

// get the timers for its action

int& restingTime();

int& shoutResting();

int& numSquares();

Reason:

These are functions that enables the derived classes of Protesters to access/change the private variables (resting tick timer, shout resting timer, and number of squares moved in current direction) of Protester.

// update this upcoming turn by changing the lastperpturn timer

void upDateLastPerp(GraphObject::Direction d);

Reason:

Protester will make a perpendicular turn if it didn't do that for 200 or more ticks. This functions will: 1 record the time if this turn is perpendicular. 2 increment the time variable if this turn is not perpendicular.

// how long since last perpendicular turn

int getLastPerp();

private:

// state if it is leaving the field

bool leaveField;

// resting ticks

int restingTimer;

// shout resting ticks

int shoutRestingTimer;

// num Squares Remain Current Direction

int numSquaresRemain;

// havent turn perp

int lastperpturn;

};

Two derived Protesters: Regular and HardCore

These two classes are the final specialization toward a real Regular / HardCore protester.

class Regular: public Protester{

public:

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

~Regular();

void becomeRich();

Reason:

When a DGold object sees a regular protester in its distance of 3, it will call this becomeRich() method to set this protester's state to leaveTheField. This method also increases the score of player by using getWorld()->increaseScore().

void doSomething();

// 1 check alive

// 2 if resting

// move and update

// 3 if leave the field

// search way to exit, and setdead after

// 4 shout to tunnelman

// 5 if can see tunnelman: turn

// else go to a random direction if it finishes its numSquareToMove

// 6 if can't go to this direction: set numSquareToMove to 0

// 7 move its position

};

class HardCore: public Protester{

public:

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

~HardCore();

void becomeRich();

Reason:

When a DGold object sees a hard-core protester in its distance of 3, it will call this becomeRich() method to increase the restingTimer of this protester to let it wait for a few more ticks. This method also increases the score of player by using getWorld()->increaseScore().

void doSomething();

Reason:

Almost the same as regular, but between checking if need to shout to tunnelman and checking if it need to leave the field (between 3 and 4 in regular protester). This hardcore protester will use the maze finding function Direction dirTo(int x, int y, int& moves); method that is declared as protected in Protester to see how many steps does it takes to get to the tunnelman. If they are close enough, this hard core protester will go to the direction that is returned from dirTo (which returns a direction object that this protester should go if it wants to go to the x,y position passed as argument).

};

**2. the functionality that I failed to implement**

I implemented all the required functionality. There are no functionalities that are required by the specs and not implemented in my game.

However, there is a problem that when there are too many (more than 6) hard core protesters, the game runs slower (ticks will take more time to execute) because the maze finding algorithm takes some time to get the result.

I make them get the distance with the player first to see if the distance between them is 4 < distance < 23. (23 is because ceiling((16^2 + 16^2)^(1/2)) is equal to 23, in other words, if the distance between them are more than 23, there is no need to use the maze finding function since if the protester can walk to the tunnel man, they must be more than 16 steps away from each other) This dramatically improved the game in terms of run speed by calling less times of the maze finding algorithm.

**3. test cases for each class**

TunnelMan

1. can it go out of bound?

2. can it remove earth and play the right sound?

3. can it squirt and decrement squirt count and play the right sound?

4. can it be annoyed when close to a falling boulder or a protester?

5. can it go through boulder?

6. can it use sonar to make nearby gold and oil visible?

7. can it drop gold nuggets and decrement gold count?

8. can it go near a barrel or gold to make it visible?

9. can it die and lose a life?

Regular Protesters

1. can it go out of bound?

2. can it remove earth?

3. can it be annoyed when close to a falling boulder

4. can it go through boulder?

5. can it shout when there is a tunnelman near it?

6. can it make a perpendicular turn after a while?

7. can it immediately change direction to face tunnelman when it sees him?

8. can it exit the maze always using the shortest path?

9. can it be stuned and annoyed by the squirt object?

10. can it pick up only the gold dropped by the tunnelman?

Hard Core Protesters

1. go through all the test cases in Regular protesters (1-10)

2. can it go directly to the location of tunnelman if it is within 16 steps to find him?

Earth

1. can it display itself in the position specified in the constructor?

2. can it be removed by the protester?

Dummy

It is an empty actor, doesn't do anything except holding a x, y coordinate.

Squirt

1. can it travel for only 4 squares if without block?

2. can it be stopped by earth, boundary, and proteseter?

3. can it stun and annoy the protesters correctly?

4. will it not appear if the tunnelman is too close to something that blocks the squirt?

Boulder

1. can it fall after few seconds after the earth beneath it is dug away?

2. can it bonk protesters and tunnel man?

3. can it hold in place if the earth beneath it is not dug away?

4. can it disappear if it falls and hit boundary or earth objects?

5. can they play the right sound when they start to fall?

DGold

1. can they disappear after few while if it is not picked up by protesters?

2. can they make the protesters "become rich"?

3. can they picked up by tunnelman?

4. are they start-off visible?

Barrel, PGold

1. can they stay invisible until player goes near them?

2. are they pickupable to the player?

3. does the oil count and gold count change accordingly if Barrel or Gold is picked up?

4. are they created with the position fully overlapping with earth or goes out of bound?

5. can they play the right sound when they died?

WaterPool, Sonarkit

1. are they start off visible?

2. are they pickupable to the player?

3. does the squirt count and sonar count change accordingly when picked up?

4. are they created with the position not overlapping with earth or goes out of bound?

5. does sonar kit only created on top left corner?

6. can they play the right sound when they died?