Tanzeela Khan

204577214

Report: Project 3

1. **StudentWorld member functions:**

   int init() was created to initialize all the actors and game board.

   int move() creates new objects at random times and calls the doSomething() method for each actor.

    virtual void cleanUp() clears the game board for a new level. I made this function virtual because I believed this function was important at the end of every level. For the rest of my functions, I only made them virtual if other classes defined their own version of the fufnciton.

    void setDisplayText() creates the ongoing display.

    int actorsCount() returns the current amount of actors in my list.

    void incActorsCount() increases the amount of actors in my list.

    void destroyDirt(int x, int y) deletes a dirt section within the game board.

    bool isDirtBelow(int x, int y) used specifically for Boulders to decide if they should keep falling. I probably should have made this virtual for dirt.

    bool isDirtinArea(int x, int y) checks within a 4x4 area for dirt.

    bool isBoulder(int x, int y) checks for boulders within radius.

    std::vector<Actor\*> accActors() returns an actor when searching for a specific type.

    void addActortoVector(Actor\* daNewActor) adds actor to list, only studentWorld can do so.

    int randInt(int max) generates a random number up to a max, probably should have used a more efficient encoded random generator.

    int findRadius(int x1, int y1, int x2, int y2) finds the distance between objects.

    bool isRadiusTooSmall(int x, int y) uses findRadius to check if radius between objects is too small.

    bool isNearFrackman(int x, int y, bool& through) calls isRadiusTooSmall to check the distance of objects from trackman.

    bool isNearProtestor(int x, int y, Protestor\*& daProtestor) Is similar to isNearFrackMan, and probably should have been created as a template function for less repetition.

    bool isObstacle(int x, int y) calls isBoulder and isDirt functions while checking for within boundaries.

    FrackMan\* accPlayer() const is necessary to access the player which is held by the game board (student class).

    bool isActorThere(int x, int y) checks for Actor at position.

    bool canWalkStraight(int sx, int sy, int ex, int ey, int& whichDirection) checks for obstacles from positon to another in a continuous non-diagonal direction.

    GraphObject::Direction chooseRandDir() chooses a random direction using the random number generator.

    bool isProtestorFacingFrackMan(Protestor\* p) checks if protestor is facing the area of trackman.

    bool isValidDir(Protestor\* p, GraphObject::Direction dir) when choosing directions, checks if possible direction has no obstacles.

    bool canMovePerpen(Protestor\* p, GraphObject::Direction& dir) checks for obstacles within erpendicular directions.

    int getProtestorCount() returns the number of protestors.

    int getBarrelsCount() returns the number of barrels of oil.

    void incBarrelsCount(int num) increments the number of barrels.

    void incProtestorCount(int num) increments the number of protestors, used for calculating max protestors per game level.

    void illuminate(int x, int y) calls isVisible to all objects within defined radius of position.

virtual ~StudentWorld() deletes all new command created objects.

**Actors.cpp member functions:**

class Actor:

    Actor(int ID, StudentWorld\* a, int x, int y, Direction dir, double size, int depth): pure virtual base class constructs actor with defined values. Sets all to visible.

    virtual void doSomething(); creates a doSomething function that can be changed by other actors. each actor creates their own.

    virtual ~Actor() uses virtual destructors whenever have base classes. All derived destructors of this base class do nothing.

    bool isAlive(); checks to see if alive

    void kill(); sets actor’s state to dead and makes it invisible.

    StudentWorld\* getWorld() const; returns a pointer to student world.

    virtual void b() const = 0; makes Actor a pure virtual class. All derived b() functions do not do anything.

    int getTicks(); returns current amount of ticks.

    void setTicks(int num); sets the current amount of ticks

  ec);

class Agent:

    Agent(StudentWorld\* a, int x, int y, int ID, Direction dir, int hitPts);

    virtual void b() const = 0;

    void  incHitPoints (int num); increment healh count

    int getHitPoints(); get health count.

class FrackMan: public Agent

    FrackMan(StudentWorld\* a); constructor of agent

    virtual void b() const {}

increment and get functions for frackman’s goodies.

    void incGold(int num);

    int getGoldCt();

    void incSonar(int num);

    int getSonarCt();

    void incSquirt(int num);

    int getSquirtCt();

    virtual ~FrackMan();

class Protestor:

    Protestor(StudentWorld\* a, int numSquares, int ID);

    void setState(State s); sets current state of protestor.

    void shout(bool shout); sets shout to true for time being.

    bool getShout();  returns shout’s current state

    void makeRightTurn(bool rightTurn); does the same for right turns.

    bool getRightTurn();

    State getState(){return curState;};

    void getsGold();

    bool hasGold(); sets and returns whether has gold.

    int getNumSquares();

    void incNumSquares(int num);    sets and returns number of steps to take.

    int getInternalMovingTicks(); sets and returns ticks to wait for shouting mechanism.

    void setInternalMovingTicks(int num);

    int getInternalRightTicks(); sets and returns ticks to wait for perpendicular mechanism.

    void setInternalRightTicks(int num);

    virtual void b() const = 0; cannot create actor by itself.

class RegularProtestor:

    RegularProtestor(StudentWorld\* a, int numSquares); constructs regular protestor.

    virtual void doSomething();

    virtual void b() const {};

class HardcoreProtestor:

    HardcoreProtestor(StudentWorld\* a, int numSquares); constructs hardcore protestor.

    virtual void doSomething();

    virtual void b() const {};

class Dirt:

    Dirt(StudentWorld\* a, int x, int y); constructs dirt.

    virtual void doSomething();

    virtual void b() const {}

    virtual ~Dirt();

class Boulder:

    Boulder(StudentWorld\* a, int x, int y); constructs boulder.

    virtual void doSomething();

    virtual void b() const {};

    enum State {stable, waiting, falling};

    virtual ~Boulder();

class Squirt:

    Squirt(StudentWorld\* a, int x, int y, Direction dir); constructs squirt.

    virtual void doSomething();

    virtual void b() const {};

class ActivatingObject:

    ActivatingObject(StudentWorld\* a, int x, int y, int ID); constructs goodie.

    virtual void b() const = 0; cannot create goodie bye itself.

class Barrel: public ActivatingObject

    Barrel(StudentWorld\* a, int x, int y); constructs barrel.

    virtual void doSomething(); does nothing.

    virtual void b() const {};

    bool pickedUp(); returns whether goodie has been picked up.

class Gold:

    enum Picker {frackman, protestor};

    enum State {permanent, temporary};

    Gold(StudentWorld\* a, int x, int y, bool isVis, int frackOrProt, int tempTicks); constructs gold

    virtual void doSomething(); waits for frackman before killing itself.

    virtual void b() const {};

class Sonar:

    Sonar(StudentWorld\* a); constructs sonar.

    virtual void doSomething() waits for a temporary amount of ticks to exist, checks if near trackman to kill itself.

    virtual void b() const {};

class WaterPool:

    WaterPool(StudentWorld\* a, int x, int y); constructs waterpool.

    virtual void doSomething() waits for a temporary amount of ticks to exist, checks if near trackman to kill itself.

    virtual void b() const {};

1. Sometimes, my protestor will walk partially off the grid and sometimes it cannot detect that it should change direction. I was unable to make the recursive kill for Protestor to function. Insetad, my protestor calls the Actor’s kill(). My hardcore protestor doesn’t function correctly, so it just uses regular protestor’s implementation of doSomething(). My hardcore protestors seem to blur as they move. I did not enable protestor’s ability to pick up objects and store them in its count. I didn’t get around to creating new protestors at levels.
2. I found Protestor’s doSomething ambiguous to separate between the hardcore and regular protestor, because their functions had only a small difference. I was also confused about whether my Protestor was actually choosing to wait to make perpendicular directions, or whether it was supposed to follow FrackMan first. I did not understand the order of preference. My goodies functions have access to the player through the StudentWorld’s getPlayer; I used this access to change the state of the FrackMan from within their classes.
3. A description of how you tested each of your classes (1-2 paragraphs per class)

StudentWorld: by checking that it could successfully create new and delete dyanically allocated objects. I chedk that it oculd access specific actors, specifically the FrackMan/player. To see if “radius” functions or “obstacle” functions worked properly, I had to see the gameplay.

Actor: by checking that I could not create an actor by itself with the new command. I tested that other objects derived from it could access its member function by calling those within the derived classes. I cheked to make sure the actor’s kill() function worked.

Agent: by checking that I could not create an actor by itself with the new command. I tested that other objects derived from it could access its member function by calling those within the derived classes.

FrackMan: by calling its doSomething() and goodies functions to see if they functioned properly. I made sure Frackman would move and delete dirt as it walked though dirt. I checked that Frackman could not run through walls or boulders. I made sure when I pressed spaces, the FrackMan would attempt to execute its called properties. In one of these cases, I checked that FrackMan could create squirt objects.

Protestor: by checking that it could not create itself with the new command. Protestor had many access/set functions that I tested through the derived classes. Protestors had internal ticking counts specifically depending on whether it would make a right turn or shout.

RegularProtestor: by checking that it could access and Protestor’s functions, would wait a defined amount of ticks before moving, would change directions when hitting an obstacle, would not run into obstacles. I tested that the protestor would stop running through and getting stuck in obstacles. I tested that the protestor would attempt to follow me if I was in direct line of sight. I checked that the protestor would attempt to move when I was no longer in sight.

HardcoreProtestor: the same way I tested my Regular Protestor class, because their implementations are the same. I attempted to make a maze function, but ran out of time to do so.

Dirt: checking if it would display in the correct positions on the grid. I also checked to see if Dirt would destruct properly and be set to null when removed off the field. My Dirt is not part of my actors list in my StudentWorld class; it is a 2D array.

Squirt: checking if it would be created only when called by the Frackman class and would promptly delete itself after moving a certain number of steps or hitting a boundary.

Boulder: checking if it would display in the correct positions on the grid. I tested whether the boulder would move when no dirt/boundary was beneath it. I tested that the boulder would access and kill Agent objects if it came near them.

ActivatingObject: checking that I could not create an actor by itself with the new command. I tested that other objects derived from it could access its member function by calling those within the derived classes.

Gold: checking that it would be created at designated positions and would disappear whenever FrackMan was near. I checked that the score would change after it was picked up. I made sure gold had a special temporary state in which it would appear for only a certain amount of ticks before calling kill().

Sonar: checking that it would be created at designated positions and would disappear whenever frackman was near or it had finished incrementing its internal tickcount. I checked that the score would change after it was picked up.

Waterpool: checking that it would be created at designated positions and would disappear whenever FrackMan was near or it had finished incrementing its internal tickcount. I checked that the score would change after it was picked up.