#1. Class Descriptions:

// Class StudentWorld

class StudentWorld : public GameWorld

{

public:

//This enumerator type is used to specify which kind of goodie the player gets

enum ITEM\_NAME{GOLD, WATER, BARREL, SONAR};

StudentWorld(std::string assetDir)

: GameWorld(assetDir)

{

}

virtual int init();

virtual int move();

virtual void cleanUp();

int getCurrentScore() const{ return getScore(); }

int getCurrentGameLevel() const { return getLevel(); }

int getNumLivesLeft() const { return getLives(); }

// The following getters are used in displaying stats

// Get player’s health

int getCurrentHealth() const { return m\_frackman->getCurrentHealth(); }

// Get the number of squirts left

int getSquirtsLeftInSquirtGun() const { return m\_frackman->getSquirtsLeftInSquirtGun(); }

// Get the gold count

int getPlayerGoldCount() const { return m\_frackman->getGoldCount(); }

// Get the sonar charge

int getPlayerSonarChargeCount() const { return m\_frackman->getSonarChargeCount(); }

// Get barrels to pick up

int getNumberOfBarrelsRemainingToBePickedUp() const { return m\_frackman->getNumberOfBarrelsRemainingToBePickedUp(); }

// Check if dirt is visible

bool isDirtVisible(int x, int y) const { return !m\_dirts[x][y]->isDead(); }

// Set dirt element dead and invisible

void setDirtInVisible(int x, int y) {

m\_dirts[x][y]->setVisible(false);

m\_dirts[x][y]->setStatus(Dirt::DEAD);

}

// Increase inventory item count

void increaseInventory(ITEM\_NAME itme\_name, int howMuch);

// Check if palyer is within R radius around (x, y)

bool isplayerAround(int x, int y, double R);

// Let player do something

void playerDoSomething() { m\_frackman->doSomething(); }

// Check if palyer is dead or not

bool theplayerDied() const { return m\_frackman->getCurrentHealth() <= 0? true:false; }

// Check if player finished this level

bool theplayerCompletedTheCurrentLevel() const { return m\_frackman->getNumberOfBarrelsRemainingToBePickedUp() == 0; }

// Check if (x, y) with radius R occupied by boulder

bool occupiedByBoulderRadius(int x, int y, double R);

// Check if (x:x+sz, y:y+sz) occupied by dirt

bool occupiedByDirt(int x, int y, int sz);

// Check if(x:x+sz, y:y+sz) occupied by boulder

bool occupiedByBoulder(int x, int y, int sz);

// Add actor to the actor vector

void addActor(Actor\* ac) { m\_vec\_actors.push\_back(ac); }

// Remove dead bodies

void removeDeadActors();

// Set the goodies within radius R around (x, y) to visible

void setVisibleInRadius(int x, int y, double R);

// Check if occupied by agent like protesters/frackman

bool occupiedByAgent(int x, int y, int size);

// Get player’s x coordinate

int getPlayerX() { return m\_frackman->getX(); }

// Get player’s y coordinate

int getPlayerY() { return m\_frackman->getY(); }

// Get frackman annoyed by amount of damage

void getFrackmanAnnoyed(int damage);

// Check if (sx, sy) and (ex, ey) are lying on a straight line with width size

bool clearDirtInStraightLine(int sx, int sy, int ex, int ey, int size);

};

// Class Actor: as the base class for all of other classes in this project

class Actor : public GraphObject

{

public:

Actor(...){}

virtual ~Actor(){}

// Make doSomething virtual so that derived classes can do different things

virtual void doSomething() = 0;

// Get the world

StudentWorld\* getWorld(){ return m\_sw; }

// Check if coordinates (x, y) within radius R around this object’s location

bool inRadius(int x, int y, double R);

// Check if this object is dead

virtual bool isDead() = 0;

// Check if this object can block agent’s forwarding

virtual bool isBlockable() { return m\_blockable; }

// Check if this object pickup-able by frackman

virtual bool isPickUpAbleByFrackman() { return m\_pickupablebyfrackman; }

// Check if this object pickup-able by protester

virtual bool isPickUpAbleByProtester() { return m\_pickupablebyprotester; }

// CHeck if this object can actively pick up things

virtual bool canPickThingsUp() { return m\_canpickthingsup; }

};

// Class People: serves as the base class of protesters and frackman

class People : public Actor{

public:

// Define STATUS enumerator for different state of people object

enum STATUS {ALIVE, LEAVE, ANNOYED, DEAD};

People(...){}

virtual ~People() {}

// set object dead

virtual void setDead() { m\_status = DEAD; }

// Check object dead or not

virtual bool isDead() { return m\_status == DEAD; }

};

// Class Protester

// Regular Protester(RP): I don’t have time to implement hardcore protester

// Class Protester

class Protester : public People{

public:

Protester(...){ }

virtual ~Protester(){}

// RP’s dosomething

virtual void doSomething();

// Check object is resting or not

bool inRest() const { return m\_rest\_ticks % m\_tickstowaitbetweenmoves == 0? false:true; }

// Check object is leaving or not

bool isLeaving() const { return m\_leaving; }

// Check if it is facing to player

virtual bool seeAndFaceToPlayer();

// Check if shout in last 15 non-resting ticks

virtual bool shoutInLastNonRestingTicks();

// Check if frackman is on the same straight line with this protester

virtual bool alignWithFrackman();

// Check if it is away from frackman

virtual bool awayFromFrackman();

};

// Class Frackman

class Frackman : public Actor{

public:

// Frackman’s version of status

enum STATUS{ALIVE, DEAD};

Frackman(...){}

virtual ~Frackman(){}

// Frackman’s doSomething

virtual void doSomething();

// Set Frackman’s health

void setHealth(int h) { m\_hit = h; }

// Decrease health by amount of howMuch

void decreaseHealth(int howMuch) { m\_hit -= howMuch; }

// Increase Squirt charge by s

void increaseSquirtsLeftInSquirtGun(int s) { m\_squirts += s; }

// Increase gold count

void increaseGoldCount(int g) { m\_goldCounts += g; }

// Increase sonar charge

void increaseSonarChargeCount(int s) { m\_sonar += s; }

// Increase Barrels in oil field by b

void increaseNumberOfBarrelsRemainingToBePickedUp(int b) { m\_barrels += b; }

// Get current health

int getCurrentHealth() const { return m\_hit; }

// get squirts left

int getSquirtsLeftInSquirtGun() const { return m\_squirts; }

// Get gold count

int getGoldCount() const { return m\_goldCounts; }

// Get sonar charge

int getSonarChargeCount() const { return m\_sonar; }

// Get barrels to be picked up

int getNumberOfBarrelsRemainingToBePickedUp() const { return m\_barrels; }

// Set status

virtual void setStatus(STATUS st) { m\_status = st; }

// Get status

virtual STATUS getStatus() const{ return m\_status; }

// Check if dead

virtual bool isDead() { return m\_status == DEAD? true:false; }

// Fire squirt in direction dir

virtual void fireSquirt(Direction dir);

// Drop a gold

virtual void dropAGold();

};

// Class Dirt

class Dirt : public Actor{

public:

enum STATUS {ALIVE, DEAD};

Dirt(...){}

virtual ~Dirt(){}

virtual void doSomething(){}

virtual void setStatus(STATUS st) { m\_status = st; }

virtual STATUS getStatus() { return m\_status; }

virtual bool isDead() { return m\_status == DEAD? true:false; }

};

// Class Squirt

class Squirt : public Actor{

public:

enum STATUS{TEMPORARY, DEAD};

Squirt() { }

virtual ~Squirt() {}

virtual bool isDead() { return m\_status == DEAD; }

virtual void doSomething();

virtual void setStatus(STATUS st) { m\_status = st; }

};

// Class Boulder

class Boulder : public Actor{

public:

enum STATUS{STABLE, DEAD, WAITING, FALLING};

Boulder(...){}

virtual ~Boulder(){}

virtual void setStatus(STATUS st) { m\_status = st; }

virtual STATUS getStatus() const { return m\_status; }

virtual void doSomething();

virtual bool isDead() { return m\_status == DEAD? true:false; }

};

// Class ActivatingObject: this class is serving as the base class for pickup-able items

class ActivatingObject : public Actor{

public:

enum STATUS{PERMANENT, TEMPORARY, DEAD};

ActivatingObject(...){}

virtual ~ActivatingObject(){}

virtual STATUS getStatus() { return m\_status; }

virtual void setStatus(STATUS st) { m\_status = st; }

virtual bool isDead() { return m\_status == DEAD? true:false; }

virtual void doSomething();

virtual void playSound();

private:

// These two pure virtual functions are used in doSomething(), derived classes have to implement them explicitly to perform different behaviours

virtual void increaseScore() = 0;

virtual void increaseInventory() = 0;

};

// Class Barrels of Oil

class Barrel : public ActivatingObject{

public:

Barrel(...){}

virtual ~Barrel(){}

// play diffent sound for oil barrel

virtual void playSound();

private:

virtual void increaseScore();// { getWorld()->increaseScore(1000); }

virtual void increaseInventory();// { getWorld()->increaseInventory(StudentWorld::BARREL, -1); }

};

// Class Gold Nuggets: this is the gold which can be pickup by player only

class GoldNugget\_FrackMan : public ActivatingObject{

public:

GoldNugget\_FrackMan(...){}

virtual ~GoldNugget\_FrackMan(){}

private:

virtual void increaseScore();

virtual void increaseInventory();

};

// Class GoldNugget\_Protester: this is the gold which can be pick up by protester only

class GoldNugget\_Protester : public ActivatingObject{

public:

GoldNugget\_Protester(...){}

virtual ~GoldNugget\_Protester(){}

private:

virtual void increaseScore(){}

virtual void increaseInventory(){}

};

// Class Sonar Kit

class Sonar : public ActivatingObject{

public:

Sonar(...) { setVisible(true); }

virtual ~Sonar() {}

private:

virtual void increaseScore();

virtual void increaseInventory();

};

// Water Pool

class Water : public ActivatingObject{

public:

Water(...) { setVisible(true); }

virtual ~Water() {}

private:

virtual void increaseScore();

virtual void increaseInventory();

};

// Class Coord: which will be used in path finding algo

class Coord{

public:

Coord(int rr, int cc): m\_r(rr), m\_c(cc){}

int r() const{ return m\_r;}

int c() const{ return m\_c;}

private:

int m\_r;

int m\_c;

};

#2. Functionalities not implemented yet:

1. I don’t have time to implement hardcore-protester;
2. I implemented regular-protester’s partial functionality: my RP cannot be killed; my RP cannot go to exit automatically; my RP cannot be bribed; my RP cannot be annoyed by squirts.
3. My frackman cannot be killed by falling boulder(I dont have time to implement it);

#3. Assumptions:

1. my RP cannot stop at some point after yelling at my frackman;
2. water pools cannot overlap with each other

#4. Test:

1. Squirt: let player fire a squirt at differnt setup to see if it stops at dirt or rock and if it can go 4 squares without blockable items on the road;

2. Gold: I print out it’s coordinates and let player gets closer to see if it can be revealed and picked up;

3. Boulder: just let the player dig the dirt supporting it to see if it can fall down as expected;

4. Sonar: let the player pick up one to see if its chage increases and press z to see if it can detect items around.

5. Frackman: just try every possible interaction with other objects to see if the behaviour is expected or not;

6. Regular Protester: let them run for long enough time to see if they can turn at intersection or properly shout at frackman.

7. Water pool: wait long time to see if they are appearing at correct locations and let player go pick them up to see if the charge increases.

8. DIrt: let player dig around to see if there is any bugs;

9. Barrels: let player pick them up to see if they can enable me to finish current level;