**Project#3 Report**

1. A 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 FrackMan must have a blah function, and each type of actor defines their own special version of it.”

Below is the base class. I use: int ID() to identify all the objects; int dir() to translate the Direction variable into an int variable; void doSomething() is pure virtual because every object needs to define a new one; bool checkAlive() to check if it is alive; void setDead() to kill; bool moving(Direction d) is virtual because I redefine it for frackman since it is complicated; bool can\_annoy() is virtual for Boulder (is it falling so as to annoy others); bool can\_be\_annoy() is virtual for protesters to indicate if they are leaving this field; bool is\_secondHand() is virtual for gold to indicate if it is dropped by frackman; int get\_count() is to get the count, which is useful for every object; void set\_count(int c) is to set value; StudentWorld\* getWorld() is for access to the studentworld; void beHit(int damage) is virtual for protesters to decrease their health.

===============================================================================

*class Actor: public GraphObject*

*{*

*public:*

*Actor(int imageID, int startX, int startY, Direction dir, double size, unsigned int depth, StudentWorld\* world)*

*: GraphObject(imageID, startX, startY, dir, size, depth), m\_world(world), is\_alive(true),count(0) {}*

*virtual ~Actor() { }*

*virtual void doSomething() = 0;*

*virtual int ID() { return 0; }*

*int dir();*

*bool checkAlive(){ return is\_alive; }*

*void setDead(){ is\_alive = false; }*

*virtual bool moving(Direction d);*

*virtual bool can\_annoy() { return false; }*

*virtual bool can\_be\_annoy() { return true; }*

*virtual bool is\_secondHand() { return false; }*

*int get\_count() { return count; }*

*void set\_count(int c) { count = c; }*

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

*virtual void beHit(int damage) { }*

*private:*

*StudentWorld\* m\_world;*

*bool is\_alive;*

*int count;*

*};*

===============================================================================

Below is the dirt.

===============================================================================

class Dirt: public Actor

{

public:

Dirt(int startX, int startY, StudentWorld\* world)

:Actor(IID\_DIRT, startX, startY, right, 0.25, 3, world){

}

virtual ~Dirt(){}

virtual void doSomething() {}

private:

};

===============================================================================

Below is the man for frackman and bad\_man. int getHitPoints() is used to get health. m\_hit records the health points.

===============================================================================

class man : public Actor

{

public:

man(int imageID, int startX, int startY, Direction dir, StudentWorld\* world, int health)

:Actor(imageID, startX, startY, dir, 1, 0, world), m\_hit(health) {}

virtual ~man() {}

virtual int ID() { return 1; }

virtual void doSomething() {}

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

void beHit(int damage) { m\_hit -= damage; }

private:

int m\_hit;

};

===============================================================================

Below is the frackman. Functions are implemented to indicate water, sonar, gold and oil. bool moving(Direction d) is redefined to scan keys, change directions and move.

===============================================================================

class FrackMan: public man

{

public:

FrackMan(StudentWorld\* world)

:man(IID\_PLAYER, 30, 60, right, world, 10), m\_oil(0), m\_water(5), m\_sonar(1), m\_gold(0){}

virtual ~FrackMan() {};

bool moving(Direction d);

virtual void doSomething();

int get\_water() { return m\_water; }

void add\_water(int w) { m\_water += w; }

int get\_sonar() { return m\_sonar; }

void add\_sonar(int w) { m\_sonar += w; }

int get\_gold() { return m\_gold; }

void add\_gold(int w) { m\_gold += w; }

void add\_oil() { m\_oil += 1; }

int get\_oil() { return m\_oil; }

private:

int m\_water;

int m\_sonar;

int m\_gold;

int m\_oil;

};

===============================================================================

Below is the bad\_man. It is for two kinds of protesters. void prot\_will\_do() indicates what a protester will do when it is alive. Two protesters will inherit different functions inside it. Restcount is for rest/ moving state. count200 is for turning counting. Count15 is for shout counting. Bad\_man can be annoyed when its state is not leaving.

===============================================================================

class bad\_man : public man

{

public:

bad\_man(int mID, StudentWorld\* world, int h)

:man(mID, 60, 60, left, world, h), state(0), restcount(0), count200(200), count15(0), stepsToMoveInCurrentDirection(0){}

virtual ~bad\_man() {}

virtual void prot\_will\_do();

void leave\_map(int x, int y);

virtual int findFrackman(int x, int y) = 0;

virtual bool bribe(int x, int y) = 0;

int getState() { return state; }

void setState(int s) { state = s; }

int getRestCount() { return restcount; }

void setRestCount(int s) { restcount = s; }

int getCount200() { return count200; }

void setCount200(int s) { count200 = s; }

int getCount15() { return count15; }

void setCount15(int s) { count15 = s; }

virtual bool can\_be\_annoy() { return (state != 2); }

private:

int state; // 0: free moving, 1: stunned, 2: leaving, 3: resting

int restcount; // move interval

int count200; // turn left or right interval

int count15; // shout interval

int stepsToMoveInCurrentDirection;

};

===============================================================================

Below is the Protester. bool bribe(int x, int y) indicates whether it is bribed. int findFrackman(int x, int y) indicates the directions to move to find the frackman.

===============================================================================

class Protester : public bad\_man

{

public:

Protester(StudentWorld\* world) :bad\_man(IID\_PROTESTER, world, 5) {}

virtual ~Protester() {}

virtual int ID() { return 2; }

virtual int findFrackman(int x, int y);

virtual bool bribe(int x, int y);

virtual void doSomething();

private:

};

===============================================================================

Below is the Hardcore\_Protester.

===============================================================================

class Hardcore\_Protester : public bad\_man

{

public:

Hardcore\_Protester(StudentWorld\* world) :bad\_man(IID\_HARD\_CORE\_PROTESTER, world, 20) {}

virtual ~Hardcore\_Protester() {}

virtual int ID() { return 3; }

virtual int findFrackman(int x, int y);

virtual bool bribe(int x, int y);

virtual void doSomething();

private:

};

===============================================================================

Below is the boulder. bool can\_annoy() tells that it is falling.

===============================================================================

class Boulder : public Actor

{

public:

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

:Actor(IID\_BOULDER, startX, startY, down, 1.0, 1, world), canAnnoy(false){}

virtual ~Boulder() {}

virtual void doSomething();

virtual bool can\_annoy() { return canAnnoy; }

virtual int ID() { return 4; }

private:

bool canAnnoy;

};

===============================================================================

Below is the Squirt.

===============================================================================

class Squirt : public Actor

{

public:

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

: Actor(IID\_WATER\_SPURT, startX, startY, dir, 1.0, 2, world) {}

virtual ~Squirt() {}

virtual int ID() { return 5; }

virtual void doSomething();

private:

};

===============================================================================

Below is the pick, sub-base class for gold, pool, sonar and oil. bool be\_pickup() shows if it is picked up.

===============================================================================

class pick : public Actor

{

public:

pick(int imageID, int startX, int startY, StudentWorld\* world)

: Actor(imageID, startX, startY, right, 1.0, 1, world){}

virtual ~pick() {};

virtual void stayAlive();

bool be\_pickup();

};

===============================================================================

Below are the gold, pool, sonar and oil. They are very similar.

===============================================================================

class Gold : public pick

{

public:

Gold(int startX, int startY, StudentWorld\* world)

: pick(IID\_GOLD, startX, startY, world), secondHand(false){}

~Gold() {};

virtual void doSomething();

virtual int ID() { return 6; }

virtual void stayAlive();

virtual bool is\_secondHand() { return secondHand; }

void set\_secondHand(bool s) { secondHand = s; }

private:

bool secondHand;

};

class Oil : public pick

{

public:

Oil(int startX, int startY, StudentWorld\* world)

: pick(IID\_BARREL, startX, startY, world) {}

virtual ~Oil() {};

virtual int ID() { return 7; }

virtual void doSomething();

private:

};

class Pool : public pick

{

public:

Pool(int startX, int startY, StudentWorld\* world)

: pick(IID\_WATER\_POOL, startX, startY, world) {}

virtual ~Pool() {};

virtual int ID() { return 8; }

virtual void doSomething();

private:

};

class Sonar : public pick

{

public:

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

: pick(IID\_SONAR, startX, startY, world) {}

virtual ~Sonar() {};

virtual int ID() { return 9; }

virtual void doSomething();

private:

};

===============================================================================

Below is the StudentWolrd. Those functions are mainly for interaction between objects. m\_dirt[64][64] is to store the dirt. vector<Actor\*> m\_actor stores other objects but frackman. queue<SuperCood> mq is for path searching. m\_map[61][61] records the distance between (x,y) to (60, 60). And m\_map\_f[61][61] stores the distance between (x,y) to frackman.

===============================================================================

class StudentWorld : public GameWorld

{

public:

StudentWorld(std::string assetDir);

virtual ~StudentWorld();

virtual int init();

virtual int move();

virtual void cleanUp();

void setDisplayText();

void addDirt();

void digDirt(int x, int y);

void insert\_BGL();

void insert\_pool\_sonar();

void shoot(int x, int y, int d);

void drop\_gold(int x, int y);

void radar(int x, int y);

bool is\_near(int x, int y, int r\_sqr, int tx, int ty);

bool is\_wall\_block\_me(int x, int y, int d, int dist);

bool is\_dirt\_block\_me(int x, int y, int d, int dist);

bool is\_cover\_dirt(int x, int y);

bool is\_boulder\_block\_me(int x, int y, int d, int dist);

bool am\_I\_facing\_it(int x, int y, int d, int tx, int ty);

bool annoy\_by\_squirt(int x, int y);

bool annoy\_by\_boulder(int x, int y);

bool is\_near\_ID(int x, int y, int ID, int sqr);

int check\_game\_state();

void add\_protester();

void change\_frackman(std::string s, int val = 0);

bool annoyP(int x, int y, int ID, int sqr);

bool annoyF(int x, int y, int d);

int choose\_diretion(int x, int y);

bool check\_viable(int x, int y, int d);

int can\_move\_to\_frackman(int x, int y);

bool bribe(int x, int y);

int to\_the\_end(int x, int y);

int to\_the\_frackman(int x, int y);

void set\_map(int m\_map[61][61], int tx, int ty);

int hard\_near\_frack(int x, int y);

private:

FrackMan\* m\_frackman;

Dirt\* m\_dirt[64][64];

std::vector<Actor\*> m\_actor;

int tickCount;

int num\_prot;

std::queue<SuperCood> mq;

int m\_map[61][61]; // record the distance between (x,y) to (60, 60)

int m\_map\_f[61][61];

};

===============================================================================

**How I implemented the path searching algorithm:**

In order to make the game goes smoothly in a steady speed, every time-consuming function is better to be executed in every tick.

I use void set\_map(int m\_map[61][61], int tx, int ty); to get the distance array m\_map. First I push the end node ((60,60) or the frackman), the distance is 0. Then push the neighboring unexplored node that is viable into the queue. The distance is 1. Also do not forget to pop. After the queue all pops out. Then we finally get the distance map. It works in the reverse direction and works like wave propagation as is shown in the figures below. (in a 8\*8 grid)

The path is in blue.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 8 | 7 | 6 |  |  |  | 2 | 1 |
| 9 | 8 | 7 |  |  |  | 3 | 2 |
| 10 | 9 | 8 |  |  |  | 4 | 3 |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |
| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |

If we have the map, by searching the neighboring nodes of a target node, we can find the way back to the end node.

===============================================================================

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

In a small chance that when a protester is killed and returns back, it will stick in dirt.

1. 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.

In the document, we are not told to stop the protester after it shouts at a frackman. But I stopped the protester for 15 non-rest ticks.

1. A description of how you tested each of your classes (1-2 paragraphs per class)

For frackman, I test how it walks using keyboard and in every conditions.

For protesters, based on the frackman, I try to kill them all or get close to them to test the functions.

For boulders, dig out the dirt below and see if it falls and kill frackman and protesters.

For squirt, just shoot it to wall, dirt, boulder and protesters to check if it is dead or it kills.

For StudentWorld, in order to make sure the searching algorithm works. I printed the distance map in real time to check if the map is right comparing with hand calculation.