**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 Iceman must have a blah function, and each type of actor defines their own special version of it.”**

I. in class Actor: public GraphObject

Actor(StudentWorld\* sw, std::string name, int imageID, int startX, int startY, Direction dir, double size, unsigned int depth = 0):GraphObject(imageID, startX, startY, dir, size, depth){...} // Constructor for Actor

virtual void doSomething() = 0; //I chose this to be virtual because Actor is an abstract base class, so the Actor class itself does not have specific implementations for doSomething. However, all its other children classes must have this doSomething, so throwing = 0 at the end makes it abstract, and checks if the children classes’ doSomething methods are implemented.

virtual ~Actor(){} // Since this is a base class, the destructor must be made virtual so the program does not have run-time errors

bool isAlive(); // A common method used by all actor child classes, because the Studentworld class needs to constantly call this class on every actor object to see if they are alive. And if not, Studentworld must delete them.

void setDead(); // In order to update the life status of each character.

void setHitPoints(int points); // Humanoid actors all have initial hitpoints, so this helps initializes their points in each child actor class’s constructor.

void decreaseHitPoints(int byHowMany); // So that the humanoid actors can constantly update their health status.

int getHitPoints(); // In order to access health/hitpoints

void zeroHitPoints(); // If a humanoid actor is hit by boulder, they lose all hitpoints.

void ifMoveAhead(int&x, int&y); // This function changes x and y to the new location, if an actor moveahead in the current direction. This function is an intermediate helper function. The x and y it returns will be passed in to other functions.

bool moreThanXUnitsAway(Actor\* a1, Actor\* a2, int x\_units); // Checks if two actor objects are more than x\_units away.

bool lessThanOrEqualToXUnitsAway(Actor\* a1, Actor\* a2, int x\_units); // Checks if two actor objects are less than or equal to x\_units away

void moveForwardOneStep(); // makes the actor object move forward one step. Calling this function will get rid of the inconvenience of worrying about direction each time.

bool hasIceInHumanDirection(Direction dir); // This checks if there are any ice blocks in direction dir’s 4x4 grid. I.e., if a protester cannot move to dir because of ice

bool hasBlouderSpaceInHumanDirection(Direction dir); // This checks to see if there are any blocked spaces in direction dir, caused by boulders. If returns true, then no humanoid objects can move to dir at the current location they are standing

bool isEdgeInHumanDirection(Direction dir); // This checks to see if moving in direction dir is impossible because it makes a humanoid actor go out of border of the oil field

bool isAnnoyed(); // Checks if a humanoid actor is annoyed.

void makeAnnoyed(); // If a humanoid actor is supposed to be annoyed after some action, this function is called to change the annoyed status to true.

void resetAnnoyed(); // A humanoid character cannot STAY annoyed. So, after certain number of annoyed ticks, we have to reset it to false after annoyed-following actions are taken

void makeLeave(); // If a protester is fully annoyed, or if a regular protester is bribed, this function changes their m\_isleaving status to true, so they begin exiting the oil field.

bool isLeaving(); // Checks to see if a protester is in leaving state

std::string getname(); // Every actor object is initialized with a name, indicating which actor child class it comes from. This fuction returns the name

StudentWorld\* getWorld(); // returns the StudentWorld an actor is in.

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

II. class Ice: public Actor

Ice(StudentWorld\* sw, int x, int y):Actor(sw, "Ice", IID\_ICE, x, y, Actor::right, 0.25, 3){...} // Ice constructor

void doSomething(){} // An ice needs to dosomething each tick

~Ice(){} // Destructor. Does not really need to do anything since we have the clear function in student world. But since it’s required on the spec, here it is.

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

class Iceman: public Actor

Iceman(StudentWorld\* sw):Actor(sw, "Iceman", IID\_PLAYER, 30, 60, Actor::right, 1, 0){...} // Constructor, of course

void doSomething(); // an iceman needs to dosomething in studentworld’s move

void fireWater(); // makes iceman fire water gun

void addWater(int amount); // adds water to storage if iceman picks up water pool

void addGold(); // updates gold storage if iceman picks up gold

int getGold(); // returns gold number

int getWater(); // returns watershots number

void addScore(int amount); //add amount score to iceman’s score

void addSonar(); //add one more sonar to iceman’s storage

int getSonar(); // returns number of sonar

int getScore();//returns iceman’s score, so it can be displayed

int SquirtX(); // determins the x location of a squirt

int SquirtY(); // determins the y location of a squirt

bool notWithinBoulderRadius(); // checks to see if iceman will be within boulder raius or not.

~Iceman(){} //Desturctor which I don’t need, but required by the spec

From now on, will be skipping constructor, doSomething, and destructor for actor subclasses because all actor classes have them and their decriptions are pretty much the same as what I have above.

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

class Boulder: public Actor

bool checkIfStable(); //Checks if a boulder is stable. If not, enters waiting state and will be falling

bool hasBoulderBelow(); // checks if this boulder hits another boulder

bool hitStuff(); // checks if this boulder hits ice, bottom of field, or another boulder. If returns true, then the boulder is set dead

bool hitHuman(); // Checks if this boulder hits a humanoid actor.

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

class Protester: public Actor

int getStateCount(); // Counts the number of ticks that indicates whether a protester should move or not

void updateStateCount();// Updates the number of ticks that indicates whether a protester should move or not

void resetStateCount();// Resets the number of ticks that indicates whether a protester should move or not

void moveOneStepToExit(); // Makes a protester move one step closer to exit (60, 60)

bool isFacingIceman(); // checks if a protester is facing iceman

int getNoShoutTicks(); // checks how many ticks a protester hasn’t shouted

void updateNoShoutTicks(); // updates how many ticks a protester hasn’t shouted

int getNoPerpendicularTurnTicks(); // checks how many ticks a protester hasn’t made a perpendicular turn

void updateNoPerpendicularTurnTicks(); // updates how many ticks a protester hasn’t made a perpendicular turn

bool onTheSameLineAsIceman(); // checks if protester is on the same vertical or horizontal line as iceman

bool noBlocksOnPathToIceman(); // if the above is true, then checks if protester can move to iceman’s location at this moment in a straight line

void ZeroNumSquaresToMoveInCurrentDirection(); // makes the number of squares to move in current direction to zero

void DecrementNumSquaresToMoveInCurrentDirection();// increment the number of squares to move in current direction to zero

void PickANewValueForNumSquaresToMoveInCurrentDirection(); //as the name indicates

int getNumSquaresToMoveInCurrentDirection();//as the name indicates

void randomlyChangeDirection(); // change the protester’s direction to a new random one

bool cannotMoveAhead(); // checks if the protester cannot move ahead

bool isAtIntersection(); // checks if the protester is at an intersection

bool pickUpGoldIfThereIsOne(); // makes the protester picks up protester-pickup-able gold, if there is one

Direction pickAPerpendicularDirection(); // Picks a perpendicular direction that can move. If more then one, select randomly

Direction DirectionToIceman(); // Checks the protester’s direction in order to get to iceman

int AnnoyedTicks; // check how many ticks they protester has been annoyed

void moveSouth(); //makes protester move south

void moveNorth();//makes protester move north

void moveEast();//makes protester move east

void moveWest();//makes protester move west

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

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

class StudentWorld : public GameWorld

StudentWorld(std::string assetDir): GameWorld(assetDir) // constructor was given

virtual int init() // given, in order to initializes actor objects at the beginning of each game

virtual int move() // given, so during each tick, this function asks each actor to do something

virtual void cleanUp()// given, so the program clears up all remaining dynamically allocated objects at the end of each level

~StudentWorld() // Performs similar function as cleanUp

bool hasIce(int x, int y); // Checks if location (x, y) has ice

bool hasBlockAhead(int x, int y); // Checks if place (x, y) has stuff that blocks an object to move in the current direction forward

void destoryIce(int x, int y); // destroy ice at the 4X4 grid with lower left bottom’s location being (x, y)

void deleteAllIce(); // destroy all remaining ice in oil field

void addObject(Actor\* pObject); // add new actor object to student world

void addSonarOrWater(); // add either a sonar or water bool based on probablily

Actor\* accessObject(int i); // access the ith object in student world’s object vector

size\_t getVectorSize(); // returns the number of actor objects in studentworld

double radius(Actor\* a1, Actor\* a2); // checks the distance between actors pointed by a1 and a2

double radius(Actor\* a, int x, int y); // checks the distance between actor pointed by a, and location (x, y)

double radius(int x1, int y1, int x2, int y2); // checks the distance between (x1, y1) and (x2, y2)

int randomX(); // To determine location of boulders, gold, and oil

int randomY(int x); // To determine location of boulders, gold, and oil

bool gridHasIce(int x, int y); // check if a 4x4 grid with lower left bottom (x, y) has any ice

void decidePoolLoc(int& x, int& y); // To determine location of add-on ice

int oilLeft(); // returns the number of oil barrels left to be picked up in each level

void decreaseOil(); // decrease the numer of oil barrels left to be picked up in each level

void updateMazeSolver(); // mazesolve is a 2D array that stores the minimum number of steps each grid has in order to exit the maze. This function updates the 2D array as the iceman digs

int northValue(int x, int y); // returns the north grid’s value in maze solver

int southValue(int x, int y); // returns the south grid’s value in maze solver

int westValue(int x, int y); // returns the west grid’s value in maze solver

int eastValue(int x, int y); // returns the east grid’s value in maze solver

void decreaseNumProtesters(); // decreases the private counter of protesters when a protester die

Iceman\* getPlayer(); returns the pointer to iceman

std::string formStatusString(int level, int lives, int health, int quirts, int gold, int barrelsLeft, int sonar, int score); //this forms the status string to be displayed on top of the game

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

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

Everything else should all be implemented and working correctly, except:

1) When the protesters exit the oil field, even though they ARE able to follow the shorted path, but sometimes a little of their bodies may overlap with ice. This is because their location is determined by their lower left corner only, so they will walk on whatever walk-able spot (with no ice or other blocks) even though the rest of their bodies might overlap with ice

2) The regular protester does not detect the iceman’s cellphone signal right now. I did not have time to implement the function.

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

**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.**

1. The spec says the humanoid objects cannot move to a place within radius of 3 units of a boulder’s center. If a boulder’s location is (x, y), this means its LOWER LEFT CORNER is at (x, y), so the center should be (x+2, y+2). However, if I implanted my program assuming this (treating the word “center” as literally the center), the game does not behave as the sample program we are given. And when I treat center as the bottom left location, (x, y), it works just like the sample program. So I decided to treat the word “center” as being the location, which, for a 4X4-grid-sized image, is its lower left corner

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

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

class StudentWorld : public GameWorld

I have played the game many times to make sure things are added and removed correctly. And I have also compared the game’s interface with the sample game to make sure things are displayed correctly.

class Ice: public Actor

The Ice class is the very first class I created. I moved around the actors all over the place to make sure that the ice objects can be destroyed correctly. I have also compared my game with the sample game to make sure the size and metrics are all correct. I have also run the program multiple times to make sure its constructor and destructor(which is empty) are correct.

class Squirt: public Actor

I have tried different initial water amounts when in my iceman’s constructor in order to see the storage part works correctly. I have also shot the squirt gun at different rate and at all possible locations in my oil field, including the places next to the border, right in front of ice, in front of boulder, and in front of protesters at different distances. I have also compared my game with the sample game to make sure the squirt gun behaves exactly the same (there are a lot of details, for example, when the iceman’s feet are just a little above the top of the ice, there is only sound but squirt is not shown. Special cases also happen when the iceman are facing close to ice or boulder or the edge. I have tested all cases). In addition, the scores and behaviors are also updated correctly when squirt kills a protester.

class Sonar: public Actor

To test Sonar, I have played my game many times to make sure it gets added with the correct probability. The behavior is the same as the sample game. I have also tried creating different objects at specific locations, and make them invisible at first. And I have tested that when pressing Z, these objects become visible and sonar sounds are played if and only if they are within distance of 12 units from the iceman. I have also tested the storage amount is correct, i.e., picking up a sonar increases the sonar count, and using one decreases the count, and when the count is 0, cannot use sonar any more. (Pressing z does not give you anything)

class Boulder: public Actor

I have run my game multiple times to make sure the boulders are created at the correct location, i.e., to make sure they are all completely WITHIN the oil field’s grids, and the distance between any 2 grid follows the minimum radius requirement. I have also dig the ice below them and checked they do disappear at the correct location. (only if they hit an ice, a boulder, or the bottom of the oil field) In addition, the scores are also updated correctly when it hits a protester. And when the boulder hits iceman, the iceman dies, as required.

class Oil: public Actor

I have run and played my game multiple times to make sure their creation follows the minimum radius requirement and the number requirement. And when I dig them, the counts are updated correctly at the top of my game, and when the count reaches 0, the player finishes the level and go to the next level. In addition, the scores are also updated correctly.

class Gold: public Actor

I have run and played my game multiple times to make sure the gold’s creation follows the minimum radius requirement and the number requirement. And when I dig them, the counts are updated correctly at the top of my game. I have also test the 2 cases where a gold is either pickupable by iceman or by protester. When it’s pickupable by iceman, i.e., when they are first created, I have waited long enough to ensure they are in permanent state. And when a gold is dropped by iceman, I have checked that it is indeed in a temporary states. The scores are updated correctly when iceman picks up gold and when iceman bribes a protester.

class Pool: public Actor

I have built and played the game multiples times to make sure the pools are created at the correct rate (based on probability) and at the correct locations. In addition, I have checked that the iceman’s water storage is updated correctly whenever the iceman picks up a pool. I have also checked that they ARE in a temporary state, and the number of ticks they appear depends on the level.

class RegularProtester: public Protester and class HardcoreProtester: public Protester

I have played the game many times to make sure they walk correctly and yells correctly. And when they are annoyed but not completely annoyed, I ensured that they do stand for certain number of ticks. Besides the two features I wrote in Question 2, everything else should work appropriately.