Report

Actor Class:

1). Actor(World\* w, int ActorID, colors ac, int x, int y);

->Since all Actors belong to the same world, the first parameter world\* is passed to Actor class.

->All Actors also have its own unique ActorID & color when first created which is stored as m\_actorID and m\_ActorColor respectively.

->All Actors start out with a preset (x,y) coordinate as indicated by the maze with the symbom ‘@’ for NachMan and ‘$’ for all Monsters. Use GetNachManStartX(), GetNachManStartY(), GetMonsterStartX(), GetMonsterStartY() in world class to set x,y.

2). virtual ~Actor();

-> Do not do anything, Notice this must be virtual because it is the base class. It will properly call the proper destructor for all of its derived classes.

3). virtual void doSomething() = 0;

->This is a pure virtual function as all Actors will be doing something, declaring this makes Actor class a Abstract Base Class.

4). void MoveActorInCurrentDirection(bool Monster);

->Since all Actors will be moving (ie. Changing its (x,y) coordinate during each tick of the game), we can have a common function in the base class to do so. The parameter for this function checks if the moving monster is a monster or else NachMan is moving.

->If monster moving, it needs to check for WALL in m\_dir. it does not require to check for CAGEDOOR.

->If NachMan is moving, it needs to check for both WALL and

CAGEDOOR in m\_dir.

-> If condition is satisfied, it will then call the proper SetX() or SetY() to move Actor to the proper direction.

**\*\*Notice Below are Getter functions they are set to const because it should not modify any private member variables. Notice each Getter functions has its complement Setter function except getMyID() and GetCurrentTickMove() as other setter functions will set these.**

5). colors GetDisplayColor() const;

int getMyID() const;

int GetX() const;

int GetY() const;

->Since Actor class stores each of the Actor’s color, actorID, X, and Y coordinates during the game. These getter functions returns the color, actorID, X and Y values of the Actor respectively at any given point in the game. Notice they are const because it should not modify any private member variables.

6). ActorState GetActorState() const;

-> Since each Actor has a state it should be in at all time in the game except when it is not initialized, it is in “NA” state. GetActorState() returns NORMAL, VULNERABLE, MONSTERDIE, RETURNTOHOME for Monsters.

GetActorState() returns ALIVE, DEAD for NachMan.

7). Direction GetActorDirection() const;

-> Since each Actor will call its doSomething() for each tick in the game, it means it will move a certain direction. This returns the Actor’s current tick direction. The SetActorDirection(Direction dir) is called to set the current tick direction for each Actor.

8). Direction GetActor\_prev\_Direction() const;

-> This functions holds the direction of the previous tick direction for each Actor.

9). bool GetCurrentTickMove() const;

-> This Function returns true if the Actor has moved during the current tick. Else it returns false if it did not move.

**\*\*Below are all Setter functions that set the value of the private member variables they should not return anything and should NOT be set to const.**

10). void SetDisplayColor(int ID);

-> This sets the display color of each Actor given a int value.

For example, if a monster is set from NORMAL state to VULNERABLE state, the parameter 0 will be passed to SetDisplayColor and set the Monsters to LIGHTBLUE color. Integer value 1 passed as an argument means the Monster is in RETURNTOHOME state and the color is set to LIGHTGRAY.

If the Monster returns back to normal state, then their respective actorID passed to this function will reset the m\_ActorColor to the Monster’s default color.

(ie. SetDisplayColor( getMyID() ); )

11). void SetActorState(ActorState ac, bool monster);

-> The first argument of this function is the desired ActorState that is to be set for the Actor; the second argument checks the Actor is a Monster.

i). For example, if the Actor is a Monster, setting the Actor to NORMAL state will immediately call the SetDisplayColor(getMyID()); this resets the Monster back to its default Monster color.

ii). If the Monster is set to MONSTERDIE state, then it will play the PAC\_SOUND\_BIG\_EAT sound.

iii). If the Monster is set to RETURNTOHOME state, then it will change the Monster color to LIGHTGRAY.

iv). If the Actor is NachMan, it will set both NachMan’s current direction and previous tick direction to NONE. This only occurs at the beginning of the game when NachMan is created or when NachMan dies.

12). void SetActorDirection(Direction dir);

-> This sets the Actor’s current tick direction and declares that the Actor has moved during the current tick.

13). void SetActor\_prev\_Direction(Direction dir);

-> This sets the Actor’s previous tick direction; it is only called at the end of each Actor’s doSomething() function. It then resets the TickDirectionSet to false, meaning the Actor has not moved yet for the next tick.

14). void SetX(int x);

void SetY(int y);

-> This sets the Actor’s current x and y position in the maze.

Notice that only the MoveActorInCurrentDirection() will call this two function. With Exception to the FastMove() function used by monsters in RETURNTOHOME state. (See explanation in FastMove())

NachMan Class:

1). NachMan(World\* w, int x, int y);

-> NachMan is created with a World class pointer passed to it along with its starting x and y coordinate on the maze map.

-> The Constructor calls the Actor class constructor passing the world pointer, NachMan ID, NachMan Color, and starting x and y coordinate in maze map.

2). ~NachMan() //Do nothing

3). void doSomething();

-> Since Actor class has a pure virtual function doSomething() and NachMan is a type of Actor, we must define our doSomething() to work properly.

-> Attempts to get a direction from the user during each tick of the game. If given a direction by the user, it will check if that direction has a WALL or a CAGEDOOR; if no WALL or CAGEDOOR, NachMan will move to that direction.

->NachMan then calls SetActor\_prev\_Direction(GetActorDirection())

to set the current tick direction as the previous tick direction since NachMan has move during this tick.

->NachMan will then call the EatMazeXY() to remove the pellets and power pellet from the maze map.

->NachMan also check if it landed on the same (x,y) position as a Monster. Either eating the Monster if its in vulnerable state or setting NachMan to DEAD state if Monster is in Normal state.

4). void DecrementNumLives();

-> NachMan stores its life count in NachMan class since other Actor’s have infinite life. Calling this function will decrement NachMan’s life count by 1.

5). void SetNachScore(int score);

-> This function adds score to NachMan’s m\_score every time it is called.

**\*\*Below are Getter Functions and should be declared const since it should not modify any member variable.**

6). int GetScore() const

int GetNumLivesLeft() const

-> Returns the score and remaining life of the Player (ie. Nachman)

**\*\*Below are functions declared private since other class should not be able to perform the following behaviors. Also notice they are not const because they in one-way or another modifies the member variable of either the derived class or base class.**

7). void checkNachDirection(Direction dir);

-> Given a direction, this function checks if the given position is a WALL or CAGEDOOR before setting NachMan’s current tick direction to the given direction. If the given direction contains a WALL or CAGEDOOR, the current tick direction is set to the previous tick direction. Otherwise, set NachMan direction to the given direction.

8). void EatMazeXY();

-> This function uses the GetGridContents function from the world class. If the current GridContent at NachMan’s current position (x,y) is a pellet, it removes the pellet from the maze map, add 10 points to NachMan, and plays the PAC\_SOUND\_SMALL\_EAT sound.

-> If the current GridContent is a power pellet, it will remove the power pellet from maze, add 100 points to NachMan, play PAC\_SOUND\_BIG\_EAT sound, and it will set all Monsters currently in NORMAL or VULNERABLE state to vulnerable state again (notice its vulnerable tick resets as well, done so with SetToVulnState )

Monster Class:

1). Monster(World\* w, int ActorID, colors mc, int x, int y);

-> Monster is created with a World class pointer passed to it along with its actor ID, actor color and starting x and y coordinate on the maze map.

-> The Constructor calls the Actor class constructor passing the world pointer, actor ID, Monster Color, and starting x and y coordinate in maze map.

2). ~Monster() {} //do Nothing

3). virtual void doSomething();

-> This is declared virtual because each derived Monster has its own unique doSomething() under different conditions. However, all monsters have the common behavior when a Monster is isn the MONSTERDIE, RETURNTOHOME, and VULERNBERABLE state except for Clyde which is a special case.

-> Monster’s in MONSTERDIE state simply sets the Monster to RETURNTOHOME state and do not move during the current tick.

-> Monster’s in RETURNTOHOME state calls the GetNextCoordinate function to find the most optimum path to return to the Monster’s starting (x,y) position. Change Monster back to NORMAL state if Monster reaches starting (x,y) position.

(Notice it calls the fastMove function to move as oppose to normalMove function, see fastMove function for explanation)

-> Monster’s in VULNERABLE state randomly picks a target (x,y) position and move toward it. This decrements the vulnerable tick by one each time doSomething is called. When vulnerable tick equals 0, Monster is set back to NORMAL state. If Monster lands on the same (x,y) position as NachMan then it will set the Monster to MONSTERDIE state.

5). int GetVulnTicks() const;

-> Returns the remaining vulnerable count of the Monster.

6). void SetToVulnState();

-> When this function is called, it changes the Monster color to Light Blue, sets the Actor/Monster state to Vulnerable and assigned the proper Monster vulnerable ticks base on the level the user is in.

7). void DecremVulnTicks()

-> Decrement Monster vulnerable tick by one

8). void normalMove(int x, int y);

-> This function follows the algorithm described in the NachMan game specification.

-> First checks if Monster is in NORMAL or VULNERABLE state before doing anything. If so, checks if Monster can move East or West closer to the given (x,y) position, if not, it checks if Monster can move North or South closer given (x,y) position.

If Monster direction has not been set, it will try each of the 4 possible directions to move. The Monster calls checkDirection function to check if the target direction contains a WALL or if it reverses the Monster’s previous direction.

If Monster direction has not been set, then it will set to the opposite of previous tick direction.

-> Once the Monster direction is set, it will call MoveActorInCurrentDirection(true) to move the Monster in that direction. Then it will call SetActor\_prev\_Direction(GetActorDirection()) to set previous tick direction to Monster’s current tick direction.

9). void randomMove();

-> calls rand() to randomly pick a (x,y) position and pass it to normalMove(x,y)

10). void fastMove(int x, int y);

-> Checks if Monster is in RETURNTOHOME state, if so, it will call the SetX() or SetY() depending on which x or y was change from the GetNextCoordinate function.

(Notice we do not need to check for WALLs because the GetNextCoordinate always checked for it).

**\*\*Below are functions declared private since other class should not be able to perform the following behaviors. Also notice they are not const because they in one-way or another modifies the member variable of either the derived class or base class.**

**Except for EastWestCol and NorthSouthRow function.**

11). void sameNachManXY(ActorState ms);

-> If Monster is in same position as NachMan, it will set NachMan to DEAD if Monster is in normal state. If Monster is in vulnerable state, it will set Monster to MONSTERDIE state and give NachMan 1000 points.

12). void checkDirection(const Direction dir);

-> This function Checks if the given function contains a WALL or a CAGEDOOR, if it does not, then the Monster will set its direction to the given direction by calling the SetActorDirection function.

13). Direction EastWestCol(int x, int y) const;

-> Given a x and y target, this function checks if it is in the same column, if so, it will return NORTH if current Monster position y is greater than target y position, SOUTH otherwise.

If not in same column, the function returns WEST if current Monster position x is greater than target x position, EAST otherwise.

14). Direction NorthSouthRow(int x, int y) const;

-> Given a x and y target, this function checks if it is in the same row, if so, it will return WEST if current Monster position x is greater than target x position, EAST otherwise.

If not in same row, the function returns NORTH if current Monster position y is greater than target y position, SOUTH otherwise.

Clyde Class:

1). Clyde(World\* w, int x, int y);

-> Clyde class is created with world pointer, x, and y value passed to it. It calls the Monster class constructor passing

Monster(w, ITEM\_MONSTER4, LIGHTCYAN, x, y).

2). virtual ~Clyde() {} //do nothing

3). void doSomething();

-> If Clyde is in NORMAL state, it will randomly pick a target (x,y) position to move and check if it landed on same position as NachMan.

-> If Clyde is in VULNERABLE state, it will call find\_n\_setNachManQuadrant function to set a target (x,y) to the diagonal quadrant of where NachMan is current at. It will then call normalMove to move toward the target (x,y) position.

It will then decrement the vulnerable count by 1 and check if vulnerable count equals 0 for which case it will set Clyde back to NORMAL state. At the end it will check if it landed on the same position as NachMan by calling sameNachManXY function. Then returns.

-> If Clyde is in MONSTERDIE or RETURNTOHOME state it will call Monster class’s doSomething function.

4). void find\_n\_setNachManQuadrant(int& x, int& y) const;

-> This is a private function of Clyde since it is unique to Clyde. Const means it should not modify any member variables in calling this function. Follows the NachMan Game specification.

Stinky Class:

1). Stinky(World\* w, int x, int y);

-> Stinky class is created with world pointer, x, and y value passed to it. It calls the Monster class constructor passing

Monster(w, ITEM\_MONSTER2, LIGHTGREEN, x, y).

2). virtual ~Stinky() {} //do nothing

3). void doSomething();

-> If Stinky is in NORMAL state, it will call find\_n\_setNachMan5

function returning true or false if NachMan is within 5 x and y coordinates of NachMan. If true, Stinky will move toward NachMan with NachMan’s current position passed to the normalMove function. If false, it will call randomMove function to move. At the end it checks if Stinky landed on same position as NachMan by calling sameNachManXY function.

->If Stinky is in MONSTERDIE, RETURNTOHOME, or VULNERABLE state, it will call Monster class’s doSomething function

4). bool find\_n\_setNachMan5() const;

-> Returns true if the absolute difference between Stinky’s x coordinate and NachMan’s x coordinate AND Stinky’s y coordinate and NachMan’s y coordinate is less than or equal to 5. Otherwise return false.

-> This is a private Stinky function, unique only to Stinky.

Dinky Class:

1). Dinky(World\* w, int x, int y);

-> Dinky class is created with world pointer, x, and y value passed to it. It calls the Monster class constructor passing

Monster(w, ITEM\_MONSTER3, LIGHTMAGENTA, x, y).

2). virtual ~Dinky() {} //do nothing

3). void doSomething();

-> If Dinky is in NORMAL state, it will call the find\_n\_setNachManLine. If the function returns true, Dinky will move toward NachMan with NachMan’s current position passed to the normalMove function. If false, it will call randomMove function to move. At the end it checks if Dinky landed on same position as NachMan by calling sameNachManXY function.

-> ->If Dinky is in MONSTERDIE, RETURNTOHOME, or VULNERABLE state, it will call Monster class’s doSomething function

4). bool find\_n\_setNachManLine() const;

-> Checks if Dinky is in same row or column as NachMan. If either condition is met, it will check if the direction closest to NachMan contains a WALL. It does so by a for-loop to check for any WALLs existing between NachMan and Dinky. The for-loop loops the difference of NachMan’s x or y with Stinky’s x or y times. Returns true if there is no WALL, and returns false if there is WALL. Also returns false if Dinky and NachMan is not in same row or column.

Inky Class:

1). Inky(World\* w, int x, int y);

-> Dinky class is created with world pointer, x, and y value passed to it. It calls the Monster class constructor passing

Monster(w, ITEM\_MONSTER1, LIGHTRED, x, y).

2). virtual ~Inky() {} //do nothing

3). void doSomething();

-> If Inky is in Normal state, it will call InkyDecide function if previous Inky state and current Inky state does not equal.

Inky have two counters: CruiseCounter and ChaseCounter. If ChaseCounter is greater than 0, it will move toward NachMan with NachMan’s current position passed to the normalMove function. If CruiseCounter is greater than 0, it will call randomMove function to move. CruiseCounter or ChaseCounter increment by one during each tick depending which condition is met.

At the end it checks if Inky landed on same position as NachMan by calling sameNachManXY function.

-> If Inky is in MONSTERDIE, RETURNTOHOME, or VULNERABLE state, it will call Monster class’s doSomething function

-> set m\_prevState equal Inky current state.

4). void InkyDecide();

-> Set CruiseCounter and ChaseCounter to 0; Call rand() % 100, if result <= 80, set ChaseCounter =1, else set CruiseCounter =1.

Class MyWorld:

1). virtual GameStatus RunLevel();

-> Initialize NachMan, set NachStateMan to ALIVE by calling

SetActorState(ALIVE, false). Initialize Monster by using a Actor pointer (Monster pointer works as well but we can make use of polymorphism here), for each Monster, we set its state to NORMAL by calling SetActorState(NORMAL, true).

-> While NachMan did not die, or there is still remaining food in the maze, have NachMan call doSomething and if NachMan did not die, have each of the 4 Monster’s doSomething.

If there are no more food in left in the maze, goto next level.

If NachMan died, return PLAYER\_DIED.

Class MyMaze:

1). bool LoadMaze(const std::string &MazeFile);

-> If Maze::LoadMaze(MazeFile) return true, call createShortestDistance function to create a m\_dist grid of the current maze map. If return false, return false.

2). bool GetNextCoordinate(int nCurX, int nCurY, int &nNextX, int &nNextY);

-> With the m\_dist grid created, set the value current to equal the value of m\_dist[nCurY][nCurX]. If current equals then return false. Otherwise, check direction from North to East to South to West (note this is simply my own preference). If at any of the given direction, the m-dist value at that direction is equal current-1, then adjust the nNextX or nNextY accordingly.

3). Class Coord

-> createShortDistance makes use of this.

4). void createShortestDistance(int sr, int sc, int m\_dist[][MAZE\_WIDTH]);

-> Use code from HW#2, change remove the first m\_maze[][10] argument as we can get access to the maze using the GetGridContents function from world class. Change the m\_dist array to m\_dist[][MAZE\_WIDTH].

Testing for each Class:

Class Actor:

\*Since Actor class is a abstract base class, if the derived class functions properly then Actor Class is correctly implemented.

Class Monster:

\*Although Monster is not a abstract base class, if the derived class functions properly then Monster Class is correctly implemented.

Class NachMan:

For Part 1 of the Project, where MyWorld::RunLevel() only initialized NachMan object, I tested NachMan by pressing any random direction to see if NachMan continues moving in that direction. Additionally, while NachMan is moving and the WEST direction is barricaded with WALLs, I press LEFT\_ARROW to see if NachMan will change its direction and continue moving in the previous proper direction. Check if moving on a position of Pellet or Power Pellet plays the proper sound and is removed from the map.

**\*\* In MyWorld::RunLevel() change the Monster doSomething loop to only make the Monster you are testing doSomething. DO NOT CALL doSomething() FOR ALL MONSTERS!**

Class Clyde:

Now with a Moveable NachMan and Monster Clyde, move around to see if Clyde’s movement is random (ie. It is not set to target NachMan, it should not chase NachMan for more than 4-5 ticks, if it continuously chases NachMan then there is something wrong with Clyde’s doSomething function). Now eat a Power Pellet and stop, observe to see if Clyde’s movement is guided in the diagonal quadrant of where NachMan is located.

Class Stinky:

Now, test NachMan with Stinky, intentionally move near Stinky without being eaten. Observe if Stinky begins to chases NachMan. Then move as far away from Stinky as possible and observer to see if Stinky is moving randomly. I also tested this by downloading a short .wav sound and added to the SoundFX class, when getting NachMan is within 5 x and y positions of Stinky, this sound should keep playing. And this sound should not play if NachMan is far away from Stinky.

Class Dinky:

Now, test NachMan with Dinky, intentionally move near Dinky without being eaten. Observe if Dinky begins to chases NachMan. Then move away from Stinky so that NachMan is either in the same row or column as Dinky but there exists a WALL between NachMan and Dinky. Observe if Dinky begins to chase NachMan. I also tested this by downloading a short .wav sound and added to the SoundFX class, when getting NachMan is a condition to satisfy Dinky’s find\_n\_setNachManLine() function, this sound should keep playing. And this sound should not play if the condition is not satisfied.

Class Inky:

Now, test NachMan with Inky. Testing this by downloading two short .wav sound and added to the SoundFX class, Inky calls the InkyDecide(), it should play one wav sound when Inky starts to chase NachMan and it should play the other wav sound when Inky starts to not chase NachMan. Only play the sound each time InkyDecide function is called, if the sound is played too often then InkyDecide function is called too often (that means there is something wrong). Count 10 tick movements and see if the sound plays agains.

Test For all 4 Monsters/MyMaze Class/MyWorld Class

Check if eating power pellet changes all the monster color to LIGHTBLUE. Have NachMan eat a monster and observe if it goes back to Monster start position and return to normal default color. Observe to see if Monster is moving in the fastest path back to the Monster start position. When Monster is at RETURNTOHOME, check to see if it is LIGHTGRAY color. When Monsters is in VULNERABLE state check to see if they are moving randomly; for Clyde, he should be moving in the diagonal quadrant of where NachMan is located.

Eat all Pellets and Power Pellets and see if the message

"You've completed the level! Press enter to continue." displays.

Play next map, to check if Monsters in RETURNTOHOME state is moving correctly. If NachMan dies, it should display

"You died but have more lives! Press enter to continue.",

If NachMan has no more life it should display

"You have lost all of your lives! Press enter to quit.".