Ryan Lam

Project 3 Report

HIGH LEVEL DESCRIPTIONS

Class StudentWorld

StudentWorld(string assetPath): constructs StudentWorld

~StudentWorld(): destructs StudentWorld by calling cleanup()

virtual int init(): initializes class and builds the current level using level(). This is virtual because it is a redefinition of Gameworld’s init()

virtual int move(): allow each actor to doSomething(), delete dead actors, and update the status bar. This is virtual because it is a redefinition of Gameworld’s move()

virtual void cleanUp(): delete all dynamically allocated actors. This is virtual because it is a redefinition of Gameworld’s cleanup().

void level(): retrieves a level text file from an access path and dynamically allocates all default actors in the level.

bool blocked(double playerX, double playerY, Actor\* a): determines whether an actor at coordinates playerX and playerY is blocked from moving by any other existing actor

bool overlaps(double x, double y): determines if any existing actor overlaps the actor at coordinates x, y.

void activateOnAppropriateActor(Actor\* a): determines which actors overlap on actor a and calls activateIfAppropriate() on those actors.

void levelFinished(): change the current game status to a finished level

void decCitizen(): decrement numCitizens member variable

bool noMoreCitizens(): returns whether there are any remaining citizens in level

void addFuel(): add 5 fuel charges to Penelope

void addVaccine(): add 1 vaccine to Penelope

void addMine(): add 2 landmines to Penelope

void createLandmine(): pushes a dynamically allocated landmine at Penelope’s location into actor list

void createVaccine(double x, double y): pushes a dynamically allocated vaccine goodie at coordinates x, y into actor list

void createPit(double x, double y): pushes a dynamically allocated pit at coordinates x, y into actor list

void createFlame(double x, double y, Direction dir): pushes a dynamically allocated flame at coordinates x, y, with direction dir into actor list

void createVomit(double x, double y): pushes a dynamically allocated vomit at coordinates x, y into actor list

void createZombie(double x, double y): pushes a dynamically allocated zombie at coordinates x, y into actor list. Zombie has a 3/10 chance of being smart and 7/10 chance of being dumb

bool isFlameBlockedAt(double x, double y): returns whether a flame is blocked from being created at coordinates x, y

void useFlamethrower(Direction dir): given a direction, create up to 3 flames in a row in that direction as long as nothing blocks the flame

bool locateNearestCitizenTrigger(double x, double y, double& otherX, double& otherY, double& distP, double& distZ, bool& isThreat): locate the nearest zombie or Penelope to a citizen at coordinates x, y

bool locateNearestCitizenThreat(double x, double y, double& distZ): locate nearest zombie to citizen at coordinates x, y

bool locateNearestVomitTrigger(double& targetX, double& targetY, Actor\* a): locate nearest citizen or Penelope that causes a zombie at coordinates getX() and getY() to vomit onto location targetX, targetY

Actor\* locateNearestTarget(int dist, Actor\* a): calculate distance from zombie A to nearest citizen or Penelope

string getString(int num): return the string form of an ostringstream input

class Actor: public GraphObject

Actor(int imageID, double startX, double startY, Direction dir, int depth, StudentWorld\* world): construct Actor class, initializing its base GraphObject class

~Actor(): destructor for Actor, does nothing because Actor contains no dynamically allocated memory

virtual void doSomething() = 0: pure virtual function, every actor must doSomething() during a tick, even if that means nothing

virtual bool canExit(): return that an Actor cannot exit, is virtual function because derived classes from Actor are able to exit

bool isDead(): return whether actor is dead

void setDead(): set dead member Boolean to true, making an actor dead

virtual void getInfected(): infects an actor, but actually does nothing, is virtual because derived classes can be infected

StudentWorld\* getStudentWorld(): return pointer to this actor’s respective Studentworld

virtual bool canGetGoodies(): actor cannot receive goodies, virtual function because some derived classes can receive goodies

virtual bool canBeActivated(): actor cannot be activated, virtual function because some derived classes can be activated

virtual bool canInfect(): actor cannot infect others, virtual function because derived zombie classes can infect others

virtual bool blocksMovement(): actor cannot block other actors’ movement, virtual function because derived classes can block actors from moving

virtual void activateIfAppropriate(Actor\* a): does nothing, virtual function because some derived classes can be activated on other actors

virtual bool canDieFromFlame(): actor can die from flames, virtual function because derived classes cannot die from flames

virtual bool canDieFromPit(): actor cannot die from falling in pit, virtual function because derived classes can die from pits

virtual bool blocksFlame(): actor cannot block flames, virtual function because some derived classes can block flames

virtual bool canBeInfected(): actor cannot be infected, virtual function because some derived classes can block flames

virtual bool canExplode(): actor cannot explode, virtual function because derived landmine class can explode

virtual void explode(): explode method which does nothing because Actor cannot explode. Is virtual because derived class performs actions when exploding

virtual void useExitIfAppropriate(): exit method which does nothing because Actor cannot exit. Is virtual because derived class performs actions when exiting level

virtual void dieFromFlameIfAppropriate(): death by flame method which does nothing because Actor cannot die from flames. Is virtual because derived classes perform actions when dying from flames

virtual void dieFromPitIfAppropriate(): death by pit method which does nothing because Actor cannot die from pit. Is virtual because derived classes perform actions when falling into pits

virtual bool isInfected(): return that an actor is not infected

class Wall: public Actor

Wall(int imageID, double startX, double startY, StudentWorld\* world): construct Wall class and initialize base Actor class

virtual void doSomething(): does nothing because Walls cannot perform any actions. Redefined function from Actor

virtual bool blocksMovement(): returns that walls can block actor movements. Redefined Actor function

virtual bool blocksFlame(): returns that walls can block flames. Redefined Actor function

virtual bool canDieFromFlame(): returns that walls cannot die from flames. Redefined Actor function

class ActivatingObject: public Actor

ActivatingObject(int imageID, double startX, double startY, Direction dir, int depth, StudentWorld\* world): constructs ActivatingObject, initializing base Actor class

virtual void doSomething() = 0: is a pure virtual function because every derived class performs different actions and redefines doSomething()

virtual bool canBeActivated(): return that ActivatingObject can be activated

virtual bool canExit(): return that ActivatingObject cannot exit level

class Exit: public ActivatingObject

Exit(int imageID, double startX, double startY, StudentWorld\* world): constructs Exit class, initializes base ActivatingObject class

virtual void doSomething(): call world’s activateOnAppropriateActor() function. Redefined Actor function

virtual void activateIfAppropriate(Actor\* a): if actor A can block movements, call its useExitIfAppropriate() function to cause it to exit the level. Redefined Actor function

virtual bool blocksFlame(): return that exit can block flames. Redefined Actor function

virtual bool canDieFromFlame(): return that exit cannot die from flames. Redefined Actor function

class Pit: public ActivatingObject

Pit(int imageID, double startX, double startY, StudentWorld\* world): constructs Pit, initializes base ActivatingObject class

virtual void doSomething(): call world’s activateOnAppropriateActor() function. Redefined Actor function

virtual void activateIfAppropriate(Actor\* a): if actor A can die from pits, call A’s dieFromPitIfAppropriate() function. Redefined Actor function

virtual bool canDieFromFlame(): return that pit cannot die from flames. Redefined Actor function

class Flame: public ActivatingObject

Flame(int imageID, double startX, double startY, Direction dir, StudentWorld\* world): constructs Flame, initializes base ActivatingObject class

virtual void doSomething(): if flame is not dead, call world’s activateOnAppropriateActor() function and decrement life counter. If life counter equals 0, set to dead. Redefined Actor function

virtual void activateIfAppropriate(Actor\* a): if actor A is not dead, call A’s dieFromFlameIfAppropriate() function

virtual bool canDieFromFlame(): return that flame cannot die from flame. Redefined actor function

int getLifeTicks(): return current lifespan

void decLifeTicks(): decrement lifespan

class Vomit: public ActivatingObject

Vomit(int imageID, double startX, double startY, StudentWorld\* world): constructs Vomit, initializes base ActivatingObject class

virtual void doSomething(): if vomit is not dead, call world’s activateOnAppropriateActor() function and decrement life counter. If life counter equals 0, set to dead. Redefined Actor function

virtual void activateIfAppropriate(Actor\* a): infect citizen or Penelope, playing a sound if citizen is infected. Redefined actor function

virtual bool canDieFromFlame(): return that vomit cannot die from flames. Redefined actor function

int getLifeTicks(): returns current lifespan

void decLifeTicks(): decrement lifespan

class Landmine: public ActivatingObject

Landmine(int imageID, double startX, double startY, StudentWorld\* world): constructs Landmine, initializes base ActivatingObject class

virtual void doSomething(): if mine is not dead and is not active, decrease safety time. If safety time reaches 0, become active and call world’s activateOnAppropriateActor() function. Redefined actor function

virtual void activateIfAppropriate(Actor\* a): if actor A cannot explode, explode mine and set mine to dead. Redefined actor function

virtual bool canExplode(): return that landmine can explode, Redefined actor function

bool isActive(): return whether mine is active

void decreaseSafety(): decrease safety time

int getSafetyTicks(): return current safety time

void setActive(): set mine to active

virtual void explode(): create 9 flames in area surrounding mine and a pit where mine was. Play exploding sound. Redefined actor function

virtual void dieFromFlameIfAppropriate(): if mine is not dead, explode the mine and set it to dead

class Goodie: public ActivatingObject

Goodie(int imageID, double startX, double startY, StudentWorld\* world): constructs Goodie, initializes base ActivatingObject class

virtual void doSomething(): does nothing, is redefined for specific derived goodies

virtual void dieFromFlameIfAppropriate(): if goodie is not dead, set it to dead

void pickUp(Actor\* a): let Penelope pick up goodie by setting it to dead, rewarding 50 points, and playing a sound

class GasCanGoodie: public Goodie

GasCanGoodie(int imageID, double startX, double startY, StudentWorld\* world): constructs GasCanGoodie, initializes base Goodie class

virtual void doSomething(): if gas can is not dead, call world’s activateOnAppropriateActor()

virtual void activateIfAppropriate(Actor\* a): if actor A can get goodies, pick up goodie and add fuel to Penelope

class VaccineGoodie: public Goodie

VaccineGoodie(int imageID, double startX, double startY, StudentWorld\* world): constructs VaccineGoodie, initializes base Goodie class

virtual void doSomething(): if vaccine is not dead, call world’s activateOnAppropriateActor()

virtual void activateIfAppropriate(Actor\* a): if actor A can get goodies, pick up goodie and add vaccine to Penelope

class LandmineGoodie: public Goodie

LandmineGoodie(int imageID, double startX, double startY, StudentWorld\* world): constructs LandmineGoodie, initializes base Goodie class

virtual void doSomething(): if landmine is not dead, call world’s activateOnAppropriateActor()

virtual void activateIfAppropriate(Actor\* a): if actor A can get goodies, pick up goodie and add mines to Penelope

class Agent: public Actor

Agent(int imageID, double startX, double startY, StudentWorld\* world): constructs Agent, initializes base Actor class

virtual bool ableToMove(Direction dir): return false, will be redefined for specific derived classes

virtual bool blocksMovement(): return that agents can block other actor’s movements. Redefined actor function

virtual bool canDieFromPit(): return that agents can die from pits. Redefined actor function

virtual bool canExit(): return that agents can exit. Redefined actor function

class Human: public Agent

Human(int imageID, double startX, double startY, StudentWorld\* world): construct Human, initializes base Agent class

int getInfectionCount(): return current infection count

virtual bool isInfected(): return whether human is infected. Redefined actor function

void increaseInfection(): increment infection count

virtual void getInfected(): set human to infected. Redefined actor function

void clearInfection(): reset infected state to false and infection count to 0

virtual bool canBeInfected(): return that human can be infected. Redefined actor function

class Penelope: public Human

Penelope(int imageID, double startX, double startY, StudentWorld\* world): constructs Penelope, initializes base Human class

virtual void doSomething(): check if Penelope is dead, increase her infection count if infected, set her to dead if infection count reaches 500, takes in key presses corresponding to various movements and actions. Redefined actor function

virtual bool ableToMove(Direction dir): returns whether Penelope is able to move in specified direction

virtual bool canExit(): Penelope can exit level if there are no more citizens in it. Redefined actor function

void addFuel(): add 5 fuel charges

void decFuel(): decrement fuel charge

int getFuelCount()L: get current fuel charge

void addVaccine(): add 1 vaccine

int getVaccineCount(): return number of vaccines

void addMine(): add 2 landmines

void decMine(): decrement number of mines

void decVaccine(): decrement number of vaccines

int getMineCount(): return number of landmines

virtual bool canGetGoodies(): return that Penelope can get goodies. Redefined actor function

virtual void useExitIfAppropriate(): if there are no more citizens, end the level when Penelope exits. Redefined actor function

virtual void dieFromFlameIfAppropriate(): if Penelope is not dead, decrement her lives and set her to dead

virtual void dieFromPitIfAppropriate(): call dieFromFlameIfAppropriate()

class Citizen: public Human

Citizen(int imageID, double startX, double startY, StudentWorld\* world): constructs Citizen, initializes base Human class

virtual void doSomething():

checks if citizen is dead

increments infection count if infected

turns into zombie if infection count = 500

gets paralyzed every other tick

determine distance to nearest Penelope and zombie

if(Penelope is closer than zombie or no zombies in level)

move towards Penelope

if(zombie is less than 80 pixels away)

determine which of the 4 directions gets citizen furthest away from zombie

move along the best direction

virtual bool canExit(): return that citizens can exit. Redefined actor function

int getParalyze(): return current paralysis counter

void incrementParalyze(): increment paralysis counter

virtual void useExitIfAppropriate(): allow citizen to exit by decreasing number of citizens, rewarding 500 points, and setting citizen to dead. Redefined actor function

virtual void dieFromFlameIfAppropriate(): if citizen is not dead, decrease number of citizens, reward -1000 points, and set citizen to dead

virtual void dieFromPitIfAppropriate(): call dieFromFlameIfAppropriate()

class Zombie: public Agent

Zombie(int imageID, double startX, double startY, StudentWorld\* world): constructs Zombie, initializes base Agent class

virtual void doSomething(): do nothing, is redefined for specific derived classes

virtual bool canInfect(): return that zombies can infect actors

int getParalyze(): return current paralysis counter

void incrementParalyze(): increment paralysis counter

int getMovementPlan(): return current movementPlan

void setMovementPlan(int mv): set movementPlan to mv

void decrementMovementPlan(): decrement movementPlan counter

class DumbZombie: public Zombie

DumbZombie(int imageID, double startX, double startY, StudentWorld\* world): constructs DumbZombie, initializes base Zombie class

virtual void doSomething():  
check if dumbzombie is alive

paralyze every other tick

if(either Penelope or citizen in front of dumbzombie)

vomit on Penelope/citizen

if(movementPlan equals 0)

pick new random movementPlan

move according to movementPlan

virtual void dieFromFlameIfAppropriate(): randomly drop a vaccine goodie, increase score by 1000, set dumbzombie to dead. Redefined actor function

virtual void dieFromPitIfAppropriate(): if dumbzombie is not dead, increase score by 1000, set dumbzombie to dead. Redefined actor function

class SmartZombie: public Zombie

SmartZombie(int imageID, double startX, double startY, StudentWorld\* world): constructs SmartZombie, initializes base Zombie class

virtual void doSomething():

check if dumbzombie is alive

paralyze every other tick

if(either Penelope or citizen in front of smartzombie)

vomit on Penelope/citizen

if(movementPlan equals 0)

determine citizen/Penelope closest to smartzombie

set direction to get closer to citizen/Penelope

move according to movementPlan

virtual void dieFromFlameIfAppropriate(): increase score by 2000, set smartzombie to dead. Redefined actor function

virtual void dieFromPitIfAppropriate(): call dieFromFlameIfAppropriate()

CLASS TESTING

StudentWorld: I tested the Studentworld class by playing the game and testing various gameplay scenarios, since Studentworld is responsible for controlling all gameplay, including actor interactions and goodie actions. I made sure that all the goodies were created correctly and that all interactions between actors were expected.

Actor: I tested the Actor class by creating various derived classes based on Actor, since Actor is an abstract base class that cannot be instantiated. I made sure that all the virtual functions included in Actor were implemented correctly in derived classes and performed correctly in-game.

Wall: I tested the Wall class by having various actors collide with walls and making sure they were blocked correctly. I made sure that no actors could somehow get through the walls.

ActivatingObject: I tested ActivatingObject by implementing various derived classes, such as pits, exits, and goodies, since ActivatingObject is an abstract base class. I constructed a variety of different configurations of actors in my levels and made sure that they were activated under the correct conditions.

Exit: I tested Exit by making sure that Penelope and citizens overlapping with the exit caused it to perform correctly. I also ensured that other overlaps such as zombies would not cause anything to happen.

Pit: I tested the Pit class by creating various configurations of pits in my levels, both created by default and created by landmines. I made sure that citizens, zombies, and Penelope died when they overlapped with the pit, and that the pit was not affected by any other actors such as vomit or flame.

Flame: I tested flame by creating flame objects in many different scenarios, like from flamethrowers and landmines. I made sure that flames did not overlap with walls, exits, or other things that could not be destroyed by flames.

Vomit: I tested vomit by observing how zombies infected other actors. I made sure that as soon as a vomit overlapped with an actor, the actor became infected and its infection count started increasing until it became a zombie or died. I also made sure that other actors like walls were not affected by vomit in any way.

Landmine: I tested landmine by deploying landmines in different patterns and setups, such as overlapping landmines. I made sure that a landmine created flames and a pit and correctly killed actors caught in it.

Goodie: I tested goodie by instantiating various derived classes of goodies, such as vaccines and gas cans, since goodie is an abstract base class. I made sure that goodies were dynamically allocated correctly and appeared in the correct locations. I made sure that Penelope could pick up these goodies and gain the proper amount of resources.

GasCanGoodie: I tested GasCanGoodie by having Penelope collect gas cans in various situations and observing whether they incremented the fuel supply correctly. I also made sure that the gas cans were vulnerable to flames and were deallocated correctly.

VaccineGoodie: I tested VaccineGoodie by constructing various vaccine goodies around the levels and having Penelope collect them. I made sure that each vaccine incremented Penelope’s vaccine supply by only one, and that vaccines could be destroyed by flames from flamethrowers and landmines.

Agent: I tested Agent by creating various derived classes based on Agent, such as Citizen and DumbZombie, since Agent is an abstract base class. I made sure that the derived classes inherited all the correct member functions and variables.

Human: I tested Human by creating Penelope and citizens and making sure they interacted with zombies and activating objects correctly. I made sure they could be infected and reacted correctly by dying or becoming zombies.

Penelope: I tested Penelope by playing various scenarios of the game and making Penelope interact with all other actors. I made sure that the interactions were correct and Penelope moved and was blocked correctly.

Citizen: I tested citizen by creating many citizens in my levels and seeing how they interacted with other actors like flames, zombies, and exits. I made sure that they followed Penelope correctly and avoided zombies accordingly.

Zombie: I tested Zombie by creating zombies in the levels and making them interact with Penelope, citizens, and other objects. I made sure that zombies moved correctly and could vomit on humans.

DumbZombie: I tested DumbZombie by creating dumb zombies in my levels and making them interact with Penelope and citizens. I also made sure they didn’t follow Penelope or Citizens and died correctly based on pits or flames.

SmartZombie: I tested SmartZombie by creating smart zombies and making sure they followed humans and vomited on them, infecting them. I made sure they were blocked correctly and died correctly when flamed or falling into pits.