Report

1. *Public Methods and Descriptions & How I tested each class:*

**Actor**

Actor(StudentWorld\* sw, int imageID, double startX, double startY, Direction dir, int depth) – The Actor constructor uses an initializer list and extends GraphObject

doSomething() – I made this function pure virtual because an Actor class never needs to be instantiated by itself, and all classes that extend the Actor class will have to implement its specific doSomething() method. In addition, every class will have different doSomething() routines, so it’s a virtual function so that the lower classes can edit them.

isAlive() – this function checks returns the private bool that contains whether or not the actor is alive

isBlocked() – this function is virtual because certain actors that extend the Actor class are blocked, which means they cannot be overlapped by other actors.

getType() – this function returns the specific type of the actor this is (i.e. Penelope, Zombie, etc.) using enumeration. Since this function will be needed by all of the subclasses, I made it a pure virtual class so that all subclasses have to implement it. I will not go in detail for every single

getWorld() – returns the pointer to the specific world this actor is in.

overlap(Actor\* otherActor) – checks if this actor overlaps with *otherActor* the user passes in by checking the Euclidean distance from this actor to *otherActor*

setDead() – set the user’s alive state to false

distance(double x, double y) – returns a double that is calculated to be the distance from the actor’s current position to (x,y)

increaseTickCount() – increases the tickCount (or frameCount)

getTickCount() – returns the current tickCount

isEvenTick() – returns true if the current tick is an even tick, false otherwise.

*Testing Method: Since Actor is a super class of many other subclasses and cannot be instantiated by itself, the only way to test out this class is through testing other classes.*

**BlockMovement**

BlockMovement(StudentWorld\* sw, int imageID, double startX, double startY) – BlockMovement constructor uses an initializer list and extends Actor

isBlocked() – For BlockMovement, it always returns true because they cannot be overlapped by other actors.

*Testing Method: Since BlockMovement is a subclass of Actor and has a very special quality of inability to be overlapped, the way to test this class is through creating blocked objects to see if the property has been passed down.*

**Human**

Human(StudentWorld\* sw, int imageID, double startX, double startY) – Human constructor uses an initializer list and extends BlockMovement class because all Human objects cannot be overlapped. Classes that extend Human include Penelope and Citizen

int getInfectCount() const – returns the infection count

void increaseInfectCount(int num) – increase the infection count by num

void getInfected() – set infected status to true

void getCured() – set infected status to false

bool isInfected() const – returns infected status

*Testing Method: Two testing methods of this class is through testing Penelope and citizens since they extend the Human class.*

**Goodie**

Goodie(StudentWorld\* sw, int imageID, double startX, double startY) – Goodie constructor uses an initializer list and extends Actor class because other obejcts are allowed to be on top of a Goodie

virtual void doSomething() – This function is virtual because every goodie adds a different use to the inventory, so every goodie should have the ability to change the inventory however they want to.

**Citizen**

Citizen(StudentWorld\* sw, double startX, double startY) – Citizen constructor uses an initializer list and extends Human class because a citizen is a type of a human

virtual void doSomething() – citizen either chooses to run away from a zombie or run towards Penelope in this case. If it overlaps with a flame, it gets killed; if it overlaps with a vomit, it gets infected; if it overlaps with an exit, it gets saved.

**Wall**

Wall(StudentWorld\* sw, double startX, double startY) – Wall constructor uses an initializer list and extends BlockMovement because it doesn’t allow any object to overlap or go over it.

virtual void doSomething() – A wall doesn’t do much besides sitting there.

**Penelope**

Penelope(StudentWorld\* sw, double startX, double startY)

virtual void doSomething() – The Penelope’s doSomething() method gives the user to input and control the actions of Penelope. If the input is up/down/left/right, Penelope moves.

void throwFlame() – Penelope shoots 3 flames in the direction she’s facing if there are no objects blocking the flames.

void useVaccine() – Penelope uses vaccine and clears her infection count to 0

void putLandmine() – Penelope puts down a landmine.

void changeVaccine(int num) – Add num quantity of vaccine to Penelope’s inventory

void changeGas(int num) – Add num quantity of gas to Penelope’s inventory for flame

void changeLandmine(int num) – Add num quantity of Landmine to Penelope’s inventory

int getFlameCount() const – returns the quantity of flame in Penelope’s inventory.

int getVaccineCount() – returns the quantity of vaccine in Penelope’s inventory.

int getLandmineCount() const – returns the quantity of Landmine in Penelope’s inventory.

**Zombie**

Zombie(StudentWorld\* sw, double startX, double startY)

virtual void doSomething() – A zombie follows its own specific movement plan and moves through the grid and vomits on human objects if triggered.

int getMovementPlan() const – Return the current movement plan

void decrementMPlan() – decrement the current movement plan

void setMPlan(int num) – set the current movement plan to num

**Dumb Zombie**

DumbZombie(StudentWorld\* sw, double startX, double startY) – A dumb zombie’s constructor

virtual void doSomething() – The dumb zombie follows its movement plan and decrements the movement plan. It vomits at human objects if appropriate.

**Smart Zombie**

SmartZombie(StudentWorld\* sw, double startX, double startY) – A Smart zombie’s constructor

virtual void doSomething() – The smart zombie follows its movement plan and decrements the movement plan. It vomits at human objects if appropriate.

**Projectile**

Projectile(StudentWorld\* sw, int imageID, double startX, double startY) – A Projectile’s constructor

virtual void doSomething(); - The Projectile sets itself to death after 2 ticks. However, any actor that overlaps with it that is damageable is destructed in their respective doSomething() functions. This applies to both Flame and Vomit since they both destroy themselves and have basically the same doSomething() functionalities.

**Flame, Vomit (refer to Projectile)**

**Vaccine Goodie, Landmine Goodie, Gas Can Goodie**

VaccineGoodie::doSomething()

LandmineGoodie::doSomething()

GasCanGoodie::doSomething()

These functions all increment Penelope’s inventory in some way.

**StudentWorld**

void setPenelope(Penelope\* p) - set penelope to the current world's Penelope

void loadLevel(int level) - Loads the indicated level

bool doesBlockMovement(double x, double y, Actor\* actor) - checks if the current actor can move to coordinate (x,y) without being blocked given that the actor is not of type projectile

bool doesBlockFire(double x, double y) - checks if a flame can be instantiated at coordinate (x,y) without being blocked

bool doesBlockGoodie(double x, double y) - checks if a goodie can be instantiated at coordinate (x,y) without being blocked

Actor\* doesOverlapWithAnyActor(Actor\* notThisActor) - Returns the actor that the passed in actor is overlapped with, returns nullptr if there is none

void addActor(Actor\* actor) - add actor to the vector of actor pointers

double getClosestZombie(double x, double y) - get the distance to the closest zombie

Actor\* getClosestHuman(double x, double y) - get the pointer to the closest human (penelope or citizen)

int numberOfCitizensLeft() - returns the number of citizens left

void exitIsSteppedOn(bool isSteppedOn) - check if the exit object is being stepped on

bool isHumanInfrontOfZombie(double x, double y, int direction) - check if there is any Human object in the direction that the zombie is facing

1. *Failed Functionalities:*

*None*

1. *Design Decisions:*

*I used Enumerations to identify the types of Actors the world contains. This makes detecting appropriate collisions a lot easier, for example, flame destroying human and zombie objects. In addition, I did not use a ActivatingObject overarch class because it does not make much difference from just inheriting not Activating objects from the Actor class.*