Project 3 Report - Angela Kan

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:

**StudentWorld**

//The StudentWorld constructor initializes the pointer to Ghost Racer as nullptr. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

StudentWorld(std::string assetPath);

// This returns a pointer to the world's GhostRacer. Defined here to easily access the GhostRacer across classes. Non-virtual because it is the exact same behavior for each class and there are no derived classes from StudentWorld.

GhostRacer\* getPointerToGhostRacer() const;

// This adds an actor to the world by taking in a pointer to the new actor as a parameter and pushing it to the back of the list container holding all the actors. Defined here to easily add each new actor into the container. This is non-virtual because it is the exact same behavior across all classes and there are no derived classes from StudentWorld.

void addActor(Actor\* a);

// Record that a soul was saved by decrementing the data member for the number of souls to be saved for the level by 1. Defined here so other actor classes can record when a soul is saved (private data member). Non-virtual because there are no derived classes from StudentWorld, and so only one version of the function.

void recordSoulSaved();

// If actor a (holy water spray) overlaps some live actor that is affected by a holy water

// projectile, inflict a holy water spray on that actor and return true;

// otherwise if actor is not affected, return false. Defined here because only StudentWorld should be accessing its container of actors and thus only it can check to see if the holy water overlaps with any other actor. Non-virtual because there are no derived classes from StudentWorld, and so only one version of the function.

bool sprayFirstAppropriateActor(Actor\* a);

// If actor a overlaps this world's GhostRacer, return a pointer to the

// GhostRacer; otherwise, return nullptr. Defined public here in StudentWorld because that is where the overlap/interaction between actors should primarily be checked and so other actor classes can access the function. Non-virtual because there are no derived classes from StudentWorld, and so only one version of the function.

GhostRacer\* getOverlappingGhostRacer(Actor\* a) const;

//Returns pointer to collision avoidance-worthy actor in specified lane either in front of or behind (based on specified bool front) and closest to specified y position; returns nullptr if no such actor. Defined here so ZombieCab class can check for other actors by iterating through the container that only StudentWorld should access. Non-virtual because there are no derived classes from StudentWorld, and so only one version of the function.

Actor\* getClosestCAWinFrontOrBehind(int leftX, int rightX, int y, bool front);

psuedocode:

*set pointer to closest actor equal to nullptr*

*if checking in front of y:*

*if ghost racer is in specified lane/position*

*set closest actor pointer to ghost racer*

*iterate through the container of all other actors and check if actor is in specified lane/position*

*if this is the first found actor, set closet actor pointer to it*

*else compare the distances from y and set the closer one*

*return closest actor*

*if checking behind y:*

*repeat same steps for checking in front of y, but this time check for behind y as the specified position*

**Actor**

// Constructor for Actor passes arguments to GraphObject base class, then initializes all actors as initially alive. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

Actor(int imageID, double startX, double startY,

int startDirection, double size, int depth,

StudentWorld\* sw);

// Empty Destructor. Defined in case you want to add a new actor to the game that requires a different behavior for its destructor and for readability. Virtual because every actor has a different destructor called when it is removed and in the case that you want to add new actors.

virtual ~Actor(){ }

// Pure-virtual version of doSomething in Actor class because all actors in the game doSomething, and all do a different version of doSomething.

virtual void doSomething() = 0;

// Accessor for isAlive data member returns bool true if actor is alive; false if not. Defined here because all actors have alive data member/status. Non-virtual because same behavior for this function across all classes.

bool isAlive() const;

// Mutator for isAlive data member changes data member based on passed in bool argument. Defined here to mutate private data member across classes. Non-virtual because it is the same behavior for this function across all classes.

void changeAlive(bool isAlive);

// Accessor for vertical speed data member returns the vertical speed. Defined here because all actors have vertical speed data member. Non-virtual because same behavior for this function across all classes.

int getVertSpeed() const;

// Mutator for vertical speed data member sets data member to new passed in bool argument. Defined here to mutate private data member across classes. Non-virtual because it is the same behavior for this function across all classes.

void setVertSpeed(int vSpeed);

// Returns true if this object affects zombie cab placement and speed, false if not. This specific definition returns false. Defined here because every class either is or isn’t collision avoidance worthy, and to make ZombieCab’s job distinguishing between these easier. Virtual because this function returns different things for different actors/classes.

virtual bool isCollisionAW() const;

// Returns true if object can be affected by water projectile and implements the required effect on the actor. Returns false if not. Defined here to assist Spray and because every actor either can be or can’t be affected by projectile. Virtual because every class has a slightly different behavior when affected by projectile.

virtual bool beSprayedIfAppropriate();

// Returns a pointer to the StudentWorld. Defined here so the actor classes and StudentWorld can communicate/interact with each other. Non-virtual because it is the same behavior/world across all classes.

StudentWorld\* getStudentWorld() const;

// Adjust the x coordinate by dx (horizontal speed) to move to a position with a y coordinate

// determined by this actor's vertical speed relative to GhostRacser's

// vertical speed. Return true if the new position is within the view;

// otherwise, return false, with the actor set as dead. Defined here because every actor has a move implementation of this sort and for readability. Virtual because certain classes have slightly different implementations of it.

virtual bool moveRelativeToGhostRacerVerticalSpeed(double dx);

**BorderLine**

// Constructor for BorderLine passes arguments to Actor base class, then initializes vertSpeed based on the passed argument. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

BorderLine(int imageID, double startX, double startY,

int startDirection, double size, int depth,

int vertSpeed, StudentWorld\* sw);

**Agent**

// Constructor for Agent passes arguments to the Actor base class, then initializes hitPoints based on the passed argument. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

Agent(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw,

int hitPoints);

// Empty Destructor. Defined in case you want to add a new actor to the game that requires a different behavior for its destructor and for readability. Virtual because every actor has a different destructor called when it is removed and in the case that you want to add new actors.

virtual ~Agent(){ }

// Accessor for hit points data member returns the health of the agent. Defined here so all agents can access their health amount. Non-virtual because same behavior for this function across all classes.

int getHitPoints() const;

// Mutator for hit points data member sets data member to new passed in bool argument. Defined here to mutate private data member across agent classes. Non-virtual because it is the same behavior for this function across all classes.

void setHitPoints(int hp);

// Accessor for horizontal speed data member returns the horizSpeed of the agent. Defined here so all agents can access their horizontal speed. Non-virtual because it’s the same behavior for this function across all classes.

int getHorizSpeed() const;

// Mutator for hSpeed data member sets data member to new passed in bool argument. Defined here to mutate private data member across agent classes. Non-virtual because it is the same behavior for this function across all classes.

void setHorizSpeed(int hSpeed);

// Returns true because all agents affects zombie cab placement and speed. Defined here because every class either is or isn’t collision avoidance worthy, and to make ZombieCab’s job distinguishing between these easier. Virtual because this function returns different things for different actors/classes (different from base class definition).

virtual bool isCollisionAW() const;

// Move the agent. If the agent doesn't go off screen and

// should pick a new movement plan, pick a new plan. Defined pure virtual here because all agents have some implementation of this, and all the implementations for each agent are for slightly different behaviors.

virtual void moveAndPossiblyPickPlan() = 0;

// Inflicts damage (hit points decrease) on agent. Return true if this agent dies as a result and play sound, otherwise return false. Defined virtual here because all agents take damage in some form in a slightly different way.

virtual bool takeDamageAndPossiblyDie(int hp);

// Returns sound as int for when Ghost Racer gets hurt. Defined here because all agents play sound when hurt, and virtual because each sound is different.

virtual int soundWhenHurt() const;

// Returns sound as int for when Ghost Racer dies. Defined here because all agents play sound when they die, and virtual because each sound is different.

virtual int soundWhenDie() const;

**GhostRacer**

// Constructor for GhostRacer passes arguments to Agent base class, then initializes holyWaterAmount based on the passed argument. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

GhostRacer(int imageID, double startX, double startY,

int startDirection, double size,

int vertSpeed, int horizSpeed, StudentWorld\* sw,

int holyWaterAmount);

// Empty Destructor. Defined in case you want to add a new actor to the game that requires a different behavior for its destructor and for readability. Virtual because every actor has a different destructor called when it is removed and in the case that you want to add new actors.

virtual ~GhostRacer(){ }

// Has GhostRacer move, read in user input+change x/y/direction accordingly, squirt water if space is pressed, and check for collisions. Defined virtual here for Ghost Racer because it has a different behavior from the other classes that is called by other classes, and all instantiated classes need a doSomething implementation.

virtual void doSomething();

// Accessor returns number of sprays Ghost Racer has left. Defined here so other classes can access this private data member. Non-virtual because it is the same behavior for this function across all classes.

int getNumSprays() const;

// Mutator for numSprays data member updates data member by adding new passed in bool argument to it. Defined here to mutate private data member across actor classes. Non-virtual because it is the same behavior for this function across all classes.

void changeNumSprays(int waterChange);

// Has Ghost Racer spin as a result of hitting oil slick by changing its direction based on its current orientation and angle. Defined here so other classes (like OilSlick) can make GhostRacer spin. Non-virtual because there is only one implementation of the spin behavior, and only one class that spins.

void spin();

// Heals Ghost Racer by hp amount (adds it to current hp) up to 100. Defined here so other classes (like HealingGoodie) can heal GhostRacer. Non-virtual because there is only one implementation of the heal behavior, and only one class that can be healed.

void heal(int hp);

**IntelligentAgent**

// Constructor for IntelligentAgent passes arguments to Agent base class, then initializes movement plan and horizSpeed to 0. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

IntelligentAgent(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw,

int hitPoints);

//Empty Destructor. Defined in case you want to add a new actor to the game that requires a different behavior for its destructor and for readability. Virtual because every actor has a different destructor called when it is removed and in the case that you want to add new actors.

virtual ~IntelligentAgent(){ }

// Accessor returns movement plan for Intelligent Agents. Defined here so other classes can access this private data member. Non-virtual because it is the same behavior for this function across all classes.

int getMovementPlan();

// Mutator for movementPlan data member updates data member to new passed in bool argument. Defined here to mutate private data member across actor classes. Non-virtual because it is the same behavior for this function across all classes.

void setMovementPlan(int plan);

// Move the agent. If the agent doesn't go off screen and

// should pick a new movement plan, pick a new plan. Defined here because all IntelligentAgents have some similar implementation of this, and virtual because all the implementations for each agent are for slightly different behaviors.

// virtual void moveAndPossiblyPickPlan();

**HumanPedestrian**

// Constructor for HumanPedestrian passes arguments to IntelligentAgent base class, then initializes vertSpeed to -4. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

HumanPedestrian(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw,

int hitPoints);

**ZombiePedestrian**

// Constructor for ZombiePedestrian passes arguments to IntelligentAgent base class, then initializes vertSpeed to -4 and ticks to 0. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

ZombiePedestrian(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw,

int hitPoints);

**ZombieCab**

// Constructor for ZombieCab passes arguments to IntelligentAgent base class, then initializes vertSpeed to passed argument and marks damagedGhostRacer-tracker data member as false. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

ZombieCab(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw,

int hitPoints, int vertSpeed);

**Spray**

// Constructor for Spray passes arguments to Actor base class, then initializes travelDistLeft to 160. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

Spray(int imageID, double startX, double startY,

int startDirection, double size, int depth,

StudentWorld\* sw);

**GhostRacerActivatedObject**

// Constructor for GhostRacerActivatedObject passes arguments to Actor base class, then initializes vertSpeed to -4. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

GhostRacerActivatedObject(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw);

//Empty Destructor. Defined in case you want to add a new actor to game that requires a different behavior for destructor. Virtual because every actor has a different destructor called when it is removed.

virtual ~GhostRacerActivatedObject(){ }

**OilSlick**

// Constructor for OilSlick passes arguments to GhostRacerActivatedObject base class. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

OilSlick(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw);

**HealingGoodie**

// Constructor for HealingGoodie passes arguments to GhostRacerActivatedObject base class. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

HealingGoodie(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw);

**HolyWaterGoodie**

// Constructor for HolyWaterGoodie passes arguments to GhostRacerActivatedObject base class. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

HolyWaterGoodie(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw);

**SoulGoodie**

// Constructor for SoulGoodie passes arguments to GhostRacerActivatedObject base class. This is defined here because every class needs a constructor. This is non-virtual because it is a constructor.

SoulGoodie(int imageID, double startX, double startY,

int startDirection, double size,

StudentWorld\* sw);

2. As far as I know, I finished implementing all the functionality, and haven’t encountered any bugs in my classes.

3. A list of other design decisions and assumptions you made:

* I utilized the provided design discussed on the CS32 website. For example, this means for holy water projectile, I had each affected class implement their own behaviors in response to being sprayed, while I had the spray’s behavior implemented in StudentWorld as a function. Another example: for OilSlick, I had its behavior effect on Ghost Racer be implemented in GhostRacer’s functions (spin), and I just had OilSlick itself essentially call that function.
* When the spec specified the location for adding new actors as “on the road”, I used LEFT\_EDGE and RIGHT\_EDGE respectively as inclusive boundaries.
* I made the assumption that numbers included with [ ] brackets were inclusive, while ( ) was exclusive.