Raj Kumar

Professor Smallberg

Computer Science 32

27 February 2021

Project 3 Report

1. Note: When I say virtual doSomething() is in every class, I mean every class that has no derived classes. The “middle” classes that are derived from the Actors class and have their own derived classes might not have this virtual doSomething() override.

**Actors.h**

Actors Class: For every function in the Actors class, these functions were placed in this Actors class because these functions are used by every class (or almost all in the case of doSomethingBasic()), so they need to be in the base class for all of the derived classes to be inherited them, so they must be part of the Actors class.

Actor(int imageID, double startX, double startY, int dir, double size, unsigned int depth, bool startAlive, bool collisionAvoid, StudentWorld\* StudentWorld1, double verticalSpeed1, double horizontalSpeed1, GhostRacer\* GhostRacer1, bool affectedByProjectile1 = false)

:GraphObject(imageID, startX, startY, dir, size, depth), alive(startAlive), collisionAvoidance(collisionAvoid), world(StudentWorld1), verticalSpeed(verticalSpeed1), horizontalSpeed(horizontalSpeed1), racer(GhostRacer1), affectedByProjectile(affectedByProjectile1) {}//takes in inputs to initialize base class and member variables

virtual void doSomething() = 0; //this is the doSomething() function which makes the object do whatever it needs to each tick, but this depends on what the type of the object is, so this must be virtual.

void doSomethingBasic(); //this is the default moving algorithm for nearly all derived classes, this is used in only some (most) derived classes, so I just call it when its needed as part of doSomething() so no need to be virtual, I don't call it in classes that use a different moving method.

bool isAlive() { return alive; } //this just checks if something is alive or not, this is the same for every single class so doesn't need to be virtual.

void nowDead() { alive = false; } //makes the object dead, same for all derived classes so no need for virtual

bool isCollisionAvoidanceWorthy() { return collisionAvoidance; } // returns if the object isCollisionAvoidanceWorthy, which is the same for all derived classes so no need for virtual

StudentWorld\* getWorld() { return world; } // returns the base world, which again is the same for all derived classes so no need for virtual

void play(const int sound); //this is just easier syntax for the playSound(int input), which is the same for all classes so no need for virtual

double getVerticalSpeed() { return verticalSpeed; } //returns the vertical speed, which again is the same for all derived classes so no need for virtual

double setVerticalSpeed(double newSpeed) { verticalSpeed = newSpeed; return verticalSpeed; } //changes then returns the vertical speed, which again is the same for all derived classes so no need for virtual

double getHorizontalSpeed() { return horizontalSpeed; } //returns the horizontal speed, which again is the same for all derived classes so no need for virtual

double setHorizontalSpeed(double newSpeed) { horizontalSpeed = newSpeed; return horizontalSpeed; } //changes then returns the horizontal speed, which again is the same for all derived classes so no need for virtual

GhostRacer\* getRacer() { return racer; } //returns the Ghostracer pointer, which again is the same for all derived classes so no need for virtual

bool isAffectedByProjectile() { return affectedByProjectile; } ////returns the vertical speed, which again is the same for all derived classes so no need for virtual

virtual void holyWaterHit(){ }// the default is that this does absolutely nothing (as many derived classes are completely unaffected), but some derived classes are affected, so this depends on the class, so this is virtual.

BorderLine Class (derived from Actor):

BorderLine(int imageID, double startX, double startY, StudentWorld\* StudentWorld1, GhostRacer\* GhostRacer1)

:Actor(imageID, startX, startY, 0, 2.0, 2, true, false, StudentWorld1, -4, 0, GhostRacer1, false) {}//takes in inputs to initialize base class, constant inputs are the ones that the spec says that this type of object always has

virtual void doSomething() { doSomethingBasic(); } //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must do something different.

OilSlick Class (derived from Actor):

OilSlick(double startX, double startY, StudentWorld\* StudentWorld1, GhostRacer\* GhostRacer1)

:Actor(IID\_OIL\_SLICK, startX, startY, 0, randInt(2,5), 1, true, false, StudentWorld1, -4, 0, GhostRacer1, false)//takes in inputs to initialize base class and member variables, Note: random size between 2 and 5

{}

virtual void doSomething() { doSomethingBasic(); } //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

MovingActor Class (derived from Actor):

MovingActor(int imageID, double startX, double startY, int dir, double size, bool collisionAvoid, int hitpoints, StudentWorld\* StudentWorld1, int verticalSpeed1, int horizontalSpeed1, GhostRacer\* GhostRacer1, bool affectedByProjectile)

:Actor(imageID, startX, startY, dir, size, 0, true, collisionAvoid, StudentWorld1, verticalSpeed1, horizontalSpeed1, GhostRacer1, affectedByProjectile), hp(hitpoints), movementPlanDistance(0) //all Movingactors have depth 0, all actors start alive, movementPlandistance starts at 0

void takeDamage(int damage); //all MovingActors are able to take damage (which is why this function is in the MovingActor class), so this function is not virtual because this just decrements the HP by the inputted amount, and then calls nowDead() if the HP is 0 or less. This process is the same for all derived classes, so no need for virtual

int getHP() { return hp; } //returns HP, which is something that all MovingActors have (which is why this function is in the MovingActor class), so this function is not virtual because this process of getting the variable is the same for all derived classes

int setHP(int newVal) { hp = newVal; return newVal;} //sets and returns HP, which is something that all MovingActors have (which is why this function is in the MovingActor class), so this function is not virtual because this process of setting the variable is the same for all derived classes

//These three following functions are like peas in a pod. They all deal with getting, decrementing, and setting new movementPlanDist, which is used in all the derived classes of MovingActor (which is why these functions are in the MovingActor class)

//The process that these three work with are the same no matter what derived class it is, which is why none of these three are virtual

int getMovementPlanDist() { return movementPlanDistance; }

int decMovementPlanDist() { movementPlanDistance--; return movementPlanDistance; }

void setMovementPlanDist(int newPlan) { movementPlanDistance = newPlan; }

//This below is used for multiple derived classes of MovingActor(which is why this function is in the MovingActor class), so this function is not virtual because this function is just not called in derived classes that do things differently

void newMovementPlan();

GhostRacer Class (derived from MovingActor, derived from Actor):

GhostRacer(StudentWorld\* StudentWorld1)

:MovingActor(IID\_GHOST\_RACER, 128, 32, up, 4.0, true, 100, StudentWorld1, 0, 0, this, false), holyWaterCount(10) //takes in inputs to initialize base class and member variables, including speed components, position, up direction, 100 hitpoints, CAW and a few others

{}

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

void spin(); //this function spins the direction of the GhostRacer when driving over an oilslick, and this is part of the GhostRacer class because this function changes the direction of the GhostRacer object. This is not virtual because there is no equivalent spinning of any other class type

//These next three functions go hand in hand. The three functions return the holyWaterCount, which measures the amount of sprays left, or add the number of picked up holyWater sprays, or decrement the holyWaterCount when a spraying projectile is released

//Only the GhostRacer can shoot and pick up holy water, which is why these three functions are in this class. These are not virtual because there is only one class that can actually use the holy water

int getHolyWaterValue() { return holyWaterCount; }

int holyWaterPickup(int num) { holyWaterCount += num; return holyWaterCount; }

int holyWaterDecrement() { if (holyWaterCount > 0) holyWaterCount--; else return -1; }

//This function heals the GhostRacer by the healed amount, but to no more than 100 HP. Only the GhostRacer can heal, which is why this is in the GhostRacer class, and it is also not virtual for this reason (no need to be virtual if there are no more implementations of same function)

void heal(int amount) { if (getHP() + amount > 100) setHP(100); else setHP(getHP() + amount); }

Pedestrian Class (derived from MovingActor, derived from Actor):

No public methods

ZomPed Class (derived from Pedestrian, derived from MovingActor, derived from Actor):

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

//These next three functions go hand in hand. The three functions return the holyWaterCount, which measures the amount of sprays left, or add the number of picked up holyWater sprays, or decrement the holyWaterCount when a spraying projectile is released

//Only the GhostRacer can shoot and pick up holy water, which is why these three functions are in this class. These are not virtual because there is only one class that can actually use the holy water

int decTicksUntilNextGrunt() { if (ticksUntilNextGrunt > 0) ticksUntilNextGrunt--; else return -1; return ticksUntilNextGrunt; }

int getTicksUntilNextGrunt() { return ticksUntilNextGrunt; }

int setTicksUntilNextGrunt(int newTicks) { ticksUntilNextGrunt = newTicks; return newTicks; }

virtual void holyWaterHit() { gotHurt(); } //the virtual holyWaterHit() function is overrided in some classes I wrote because some classes must respond to holy water hitting them differently, while some do absolutely nothing, which is the default function in the Actor class so this must be virtual

HumPed Class (derived from Pedestrian, derived from MovingActor, derived from Actor):

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

virtual void holyWaterHit(); //the virtual holyWaterHit() function is overrided in some classes I wrote because some classes must respond to holy water hitting them differently, while some do absolutely nothing, which is the default function in the Actor class so this must be virtual

ZomCab Class (derived from MovingActor, derived from Actor):

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

virtual void holyWaterHit(); //the virtual holyWaterHit() function is overrided in some classes I wrote because some classes must respond to holy water hitting them differently, while some do absolutely nothing, which is the default function in the Actor class so this must be virtual

Projectile Class (derived from Actor):

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

void tickMoved() { travelDistLeft -= SPRITE\_HEIGHT; if (travelDistLeft <= 0) { nowDead(); }} //decrement by the sprite's height every tick, and once the travelDist is completed (when it is 0), then kill the sprite (to range limit it by 160 px). This is in the Projectile class because only projectiles have a range limit out of all classes, and for the same reason, there is no need for this to be virtual because there are no other implementations of this.

Goodie Class (derived from Actor):

Goodie(int imageID, double startX, double startY, int dir, double size, StudentWorld\* StudentWorld1, GhostRacer\* GhostRacer1, bool affectedByProjectile)

:Actor(imageID, startX, startY, dir, size, 2, true, false, StudentWorld1, -4, 0, GhostRacer1, affectedByProjectile)//takes in inputs to initialize base class and member variables, including depth of 2 and speed components

{}

HealingGoodie Class (derived from Goodie, derived from Actor):

HealingGoodie(double startX, double startY, StudentWorld\* StudentWorld1, GhostRacer\* GhostRacer1)

:Goodie(IID\_HEAL\_GOODIE, startX, startY, 0, 1.0, StudentWorld1, GhostRacer1, true) //takes in inputs to initialize base class and member variables, like direction of 0 and size of 1.0, and affectedByProjectile of true

{}

virtual void doSomething(); //the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

virtual void holyWaterHit() { nowDead(); } //the virtual holyWaterHit() function is overrided in some classes I wrote because some classes must respond to holy water hitting them differently, while some do absolutely nothing, which is the default function in the Actor class so this must be virtual

HolyWaterGoodie Class (derived from Goodie, derived from Actor):

virtual void doSomething();//the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

virtual void holyWaterHit() { nowDead(); } //the virtual holyWaterHit() function is overrided in some classes I wrote because some classes must respond to holy water hitting them differently, while some do absolutely nothing, which is the default function in the Actor class so this must be virtual

SoulGoodie Class (derived from Goodie, derived from Actor):

virtual void doSomething();//the virtual doSomething() function is in every single class that I wrote because each class must doSomething() each tick, and each class must doSomething() different, so this must be virtual

**Note: Not including the functions that had declarations given by the instructors, none of the functions I wrote myself were virtual because there are no derived classes of StudentWorld which could need different versions of these functions and because there was no need of overriding the inherited functions from GraphObject (which I don’t even think was allowed in the first place, either). Thus, there is no need to explain in the comment of each function why it was or was not virtual, because I already explained it in this note. I also don’t see the need of explaining why the 3 given functions and the destructor need to be part of StudentWorld or why they are or not virtual, as these are given and these decisions weren’t actually made by me.**

**StudentWorld.h**

virtual int init(); //this setups the road and makes the game ready to play by adding the borderlines and racer

virtual int move(); //this handles each tick and makes sure every object has a chance to move, and also deletes any dead actors every tick

virtual void cleanUp(); //this deletes all the dynamically allocated objects when you die or the game is quit

~StudentWorld() { cleanUp(); } //this helps clean up all the dynamically allocated objects at the end of the game

int savedASoul() { souls2Save--; return souls2Save; } //decrements the souls2Save variable, this is in the StudentWorld class because the StudentWorld class is the only class I wrote which is capable of keeping track number of objects of all types of classes, including the soul goodies

int fatality(); //this is to let the StudentWorld class know that the racer has died, and the "round" of the game must be ended, and the framework then can automatically call cleanUp(), this is in the StudentWorld class because you need a way to return GWSTATUS\_PLAYER\_DIED from the move() function if the player dies if, for example, a HumPed is hit

//These three functions are similar, as they are functions that allow the Actors from Actor.h and Actor.cpp to add new objects to the game either randomly when the Actor dies or when the player wants to shoot the cannon. The first function adds HealingGoodies randomly after a ZomPed is destroyed, and the second adds an oil slick randomly after a ZomCab is destroyed, and the third creates a Projectile after the space bar is pressed if there is enough holy water left

//These three functions are forced to be part of the StudentWorld class because this is the class where the vector of Actor objects lives, and thus all objects must be added somewhere through StudentWorld

void addHealingGoodieRandomly(double x\_pos, double y\_pos); //(DO NOT PUT IN THE move() function, only to be called from Actor.cpp)

void addOilSlickFromCabDeath(double x\_pos, double y\_pos); //(DO NOT PUT IN THE move() function, only to be called from Actor.cpp)

void addProjectile(double x\_pos, double y\_pos, int direction); //(DO NOT PUT IN THE move() function, only to be called from Actor.cpp)

bool overlaps(Actor\* A, Actor\* B); //This function checks if two Actors overlap by comparing their x and y coordinates and "radiuses". This must be part of the StudentWorld class because this class needs to use this function in the checkActivation(Projectile\* P) function (which is part of the StudentWorld class), which goes through all the Actors in the vector and finds the first that overlaps with itself (the Projectile)

Actor\* checkActivation(Projectile\* P); //This function goes through all the Actors in the vector and finds the first that overlaps with itself (the Projectile), excluding itself. This must be in the StudentWorld class because only a part of the StudentWorld class can traverse the private vector of Actors in the StudentWorld class

//These two functions find if the closest collisionAvoidanceWorthy Actors are within 96 px of the given ZomCab (the minInFront includes the GhostRacer, and the minBehind excludes the GhostRacer). These both must be in StudentWorld because these functions go through all the Actors in the vector to see which are the closest, and the private vector of the StudentWorld class can only be accessed through the StudentWorld class

bool minInFront(ZomCab\* ZCab);

bool minBehind(ZomCab\* ZCab);

Pseudocode:  
I will provide the pseudocode for bool minInFront(ZomCab\* ZCab), which is very similar to bool minBehind(ZomCab\* ZCab), which would have basically the same pseudocode. I would write the pseudocode for doSomething() for zombie cabs as well, but there really isn’t any point because the way the spec is written, it is pseudocode itself (just written in sentences, though). Thus, since I followed the spec very closely, I would just be rewriting the spec if I wrote the pseudocode, and there wouldn’t be any point of writing or reading that because it is basically already done in the spec. The doSomething() itself isn’t very complicated, but the minInFront and minBehind are, which is why I included one of them (as they are the same, basically). I will also provide the pseudocode for **void StudentWorld::addZomCabsRandomly()**

and corresponding helper functions bool StudentWorld::findCollisionWorthyActors(int &leftLanefromTop, int &leftLanefromBottom, int &middleLanefromTop, int &middleLanefromBottom, int &rightLanefromTop, int &rightLanefromBottom) and bool StudentWorld::addZomCab(const int leftLanefromTop, const int leftLanefromBottom, const int middleLanefromTop, const int middleLanefromBottom, const int rightLanefromTop, const int rightLanefromBottom)

**bool minInFront(ZomCab\* ZCab)**

Traverse Actor Container in Loop

a is the current Actor in loop

If(ZCab is not a and ZCab is in the same lane as current Actor and a is CAW)

True: if( Y coordinate of ZCab is less than Y coordinate of a)

If (Y coordinate of a - Y coordinate of ZCab is less than 96)

Return True

//(Loop is complete here)

Return false

**void StudentWorld::addZomCabsRandomly()**

chance is the max number of 100 minus the level times 10 and 20

if (randomNumber between 0 and chance is 0)

make six variables (you can see these written in the next line)

run findCollisionWorthyActors(leftLanefromTop, leftLanefromBottom, middleLanefromTop, middleLanefromBottom, rightLanefromTop, rightLanefromBottom)

using values stored in findCollisionWorthyActors, run bool StudentWorld::addZomCab(const int leftLanefromTop, const int leftLanefromBottom, const int middleLanefromTop, const int middleLanefromBottom, const int rightLanefromTop, const int rightLanefromBottom)

**bool StudentWorld::addZomCab(const int leftLanefromTop, const int leftLanefromBottom, const int middleLanefromTop, const int middleLanefromBottom, const int rightLanefromTop, const int rightLanefromBottom)**

make array of 3 integers

store 1,2,3 in array in random order

Repeatedly, three times, once for each lane

Current lane is the ith element in the array

If current lane is 1 and there is nothing in lane 1 or closest to bottom object is very high

True: Add zombie cab in center of lane 1 at bottom

Return true

If current lane is 1 and there is nothing in lane 1 or closest to top object is very low

True: Add zombie cab in center of lane 1 at top

Return true

Repeat checks for lanes 2 and 3

Return false //Whole function ran, and no Zombie cab was added

**bool StudentWorld::findCollisionWorthyActors(int &leftLanefromTop, int &leftLanefromBottom, int &middleLanefromTop, int &middleLanefromBottom, int &rightLanefromTop, int &rightLanefromBottom)**

Repeatedly, three times, once for each lane

find the collision worthy actors closest to top

store closest distance

find the collision worthy actors closest to bottom

store closest distance

2) I don’t have any known bugs. So long as I can see, the game I created looks just like the example, with the exception of it running much faster on my computer than the example, which the FAQ says is ok.

3) I assumed that when adding new Zombie Cabs all three lanes were supposed to be picked randomly, while the spec didn’t clearly say that the second and third lanes needed to be picked randomly.