CS32 Project 3 Report

Jerry Xu

505565699

**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. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in the Pedestrian class are able to sneeze, and each type of actor sneezes in a different way.” There's no need to write pseudocode except for the comparatively few more complex functions.**

I modeled most of my classes exactly the same as the provided project design on the website.

Here is the hierarchy:

GraphObject

Actor

BorderLine

Agent

GhostRacer

Pedestrian

HumanPedestrian

ZombiePedestrian

ZombieCab

Spray

GhostRacerActivatedObject

OilSlick

HealingGoodie

HolyWaterGoodie

SoulGoodie

This is the public interface for the Student World class:

class Student World {

// constructor/destructor

StudentWorld(std::string assetDir);

virtual ~StudentWorld();

// virtual because overrides GameWorld functions

// initialize variables, create ghost racer, create two columns of yellow borderlines, create two columns of white borderlines

virtual int init();

// virtual because overrides GameWorld functions

// Have each actor do something, including ghost racer, check if GhostRacer died or level was completed, remove all the dead actors, update last white borderline location, add new white/yellow borderlines as needed, add zombie cabs (pseudocode below), add new human pedestrians, add new zombie pedestrians, add new oil slicks, add new holy water goodies, add new soul goodies, update the Game Status display next using stringstreams, and continue the game

virtual int move();

pseudocode for adding zombie cabs:

if by chance we add a vehicle this tick:

do this up to three times:

find a random column that has not been removed

// bottom

check if ghost racer is in lane

for each actor

if it is in the current lane and is collision avoidance worthy

find the smallest y out of them

if we didn’t find any in the lane or if closest actor is too far

we found the lane to add a zombie cab

break

// top

check if ghost racer is in lane

for each actor

if it is in the current lane and is collision avoidance worthy

find the largest y out of them

if we didn’t find any in the lane or if closest actor is too far

we found the lane to add a zombie cab

break

delete the lane that we already checked

if we found a lane, add the zombie cab in that lane

// virtual because overrides GameWorld functions

// delete all actors, including ghostracer, clear the actors vector

virtual void cleanUp();

// Return a pointer to the world's GhostRacer.

// in StudentWorld because actors should only interact with other actors through studentworld

GhostRacer\* getGhostRacer();

// Add an actor to the world.

// in StudentWorld because studentworld deals with all actors

void addActor(Actor\* a);

// Record that a soul was saved.

// in StudentWorld because controls if game ends or not

void recordSoulSaved();

// returns whether any actor is close, in front, and in same lane

// used by Zombie Cabs, in StudentWorld because have to search through all actors

bool hasCloseActorFront(Actor\* a);

// return whether any actor is close, behind, and in same lane

// used by Zombie Cabs, in StudentWorld because have to search through all actors

bool hasCloseActorBack(Actor\* a);

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

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

// otherwise, return false. (See Actor::beSprayedIfAppropriate.)

// in StudentWorld because have to search through all actors

bool sprayFirstAppropriateActor(Actor\* a);

pseudocode:

for each actor

check if it overlaps with the spray actor

if it does

if it can be sprayed, then spray it

return

otherwise, no actor was sprayed and return false

// Return true if actor a1 overlaps actor a2, otherwise false.

// in StudentWorld because any actor can call it

bool overlaps(const Actor\* a1, const Actor\* a2) const;

Public interfaces for the actors:

class Actor : public GraphObject

{

public:

// constructor

Actor(StudentWorld\* sw, int imageID, double x, double y, double size, int dir, int depth);

// Action to perform for each tick.

// pure virtual because every actor has different action

virtual void doSomething() = 0;

// Is this actor dead?

bool isDead() const;

// Mark this actor as dead.

void setDead();

// Get this actor's world

StudentWorld\* world() const;

// Get this actor's vertical speed.

double getVerticalSpeed() const;

// Set this actor's vertical speed.

void setVerticalSpeed(double speed);

// get which lane it is in (left = 0, middle = 1, right = 2)

int getLane();

pseudocode:

if x value >= ROAD\_CENTER - ROAD\_WIDTH / 2 and x value < ROAD\_CENTER - ROAD\_WIDTH / 2 + ROAD\_WIDTH / 3

// left lane

return 0

else if x value >= ROAD\_CENTER - ROAD\_WIDTH / 2 + ROAD\_WIDTH / 3 and x value < ROAD\_CENTER + ROAD\_WIDTH / 2 - ROAD\_WIDTH / 3

// center lane

return 1

else if x value >= ROAD\_CENTER + ROAD\_WIDTH / 2 - ROAD\_WIDTH / 3 and x value < ROAD\_CENTER + ROAD\_WIDTH / 2

// right lane

return 2

// not on road

return -1

// If this actor is affected by holy water projectiles, then inflict that

// affect on it and return true; otherwise, return false.

// virtual because each actor has different action when sprayed

virtual bool beSprayedIfAppropriate();

// Does this object affect zombie cab placement and speed?

// virtual because not avoidance worthy unless overridden

virtual bool isCollisionAvoidanceWorthy() const;

// Adjust the x coordinate by dx 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 dead.

bool moveRelativeToGhostRacerVerticalSpeed(double dx);

};

class BorderLine : public Actor

{

public:

// constructor

BorderLine(StudentWorld\* sw, double x, double y, bool isYellow);

// Action to perform for each tick.

virtual void doSomething();

};

class Agent : public Actor

{

public:

// constructor

Agent(StudentWorld\* sw, int imageID, double x, double y, double size, int dir, int hp);

virtual bool isCollisionAvoidanceWorthy() const;

// Get hit points.

int getHP() const;

// Increase hit points by hp.

void incHP(int hp);

// Do what the spec says happens when hp units of damage is inflicted.

// Return true if this agent dies as a result, otherwise false.

virtual bool takeDamageAndPossiblyDie(int hp);

// What sound should play when this agent is damaged but does not die?

virtual int soundWhenHurt();

// What sound should play when this agent is damaged and dies?

virtual int soundWhenDie();

};

class GhostRacer : public Agent

{

public:

// constructor

GhostRacer(StudentWorld\* sw, double x, double y);

// Action to perform for each tick.

virtual void doSomething();

// return sound when dies

virtual int soundWhenDie() const;

// How many holy water projectiles does the object have?

int getNumSprays() const;

// Increase the number of holy water projectiles the object has.

void increaseSprays(int amt);

// Spin as a result of hitting an oil slick.

void spin();

};

class Pedestrian : public Agent

{

public:

// constructor

Pedestrian(StudentWorld\* sw, int imageID, double x, double y, double size);

// return sound when hurt

virtual int soundWhenHurt() const;

// return sound when dies

virtual int soundWhenDie() const;

// Get the pedestrian's horizontal speed

int getHorizSpeed() const;

// Set the pedestrian's horizontal speed

void setHorizSpeed(int s);

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

// should pick a new movement plan, pick a new plan.

void moveAndPossiblyPickPlan();

};

class HumanPedestrian : public Pedestrian

{

public:

// constructor

HumanPedestrian(StudentWorld\* sw, double x, double y);

// Action to perform for each tick.

virtual void doSomething();

// get sprayed

virtual bool beSprayedIfAppropriate();

// don't decrease own hp

virtual bool takeDamageAndPossiblyDie(int hp);

};

class ZombiePedestrian : public Pedestrian

{

public:

// constructor

ZombiePedestrian(StudentWorld\* sw, double x, double y);

// Action to perform for each tick.

virtual void doSomething();

// get sprayed

virtual bool beSprayedIfAppropriate();

};

class ZombieCab : public Agent

{

public:

// constructor

ZombieCab(StudentWorld\* sw, double x, double y);

// Action to perform for each tick.

virtual void doSomething();

// get sprayed

virtual bool beSprayedIfAppropriate();

};

class Spray : public Actor

{

public:

// constructor

Spray(StudentWorld\* sw, double x, double y, int dir);

// Action to perform for each tick.

virtual void doSomething();

};

class GhostRacerActivatedObject : public Actor

{

public:

// constructor

GhostRacerActivatedObject(StudentWorld\* sw, int imageID, double x, double y, double size, int dir);

virtual bool beSprayedIfAppropriate();

// Action to perform for each tick.

virtual void doSomething();

pseudocode:

if out of the frame

return

if overlaps with ghost racer

do the specific activity for object

play specific sound

increase the score by the specific amount

if it self destructs after touching the ghost racer

kill it

// Do the object's special activity (increase health, spin Ghostracer, etc.)

// pure virtual because each activity is completely different

virtual void doActivity(GhostRacer\* gr) = 0;

// Return the object's increase to the score when activated.

// pure virtual because each score is different

virtual int getScoreIncrease() const = 0;

// Return the sound to be played when the object is activated.

// virtual because sound is SOUND\_GOT\_GOODIE unless otherwise specified

virtual int getSound() const;

// Return whether the object dies after activation.

// virtual because most objects self destruct, but oil slicks don’t

virtual bool selfDestructs() const;

// Return whether the object is affected by a holy water projectile.

// pure virtual because roughly equal numbers of is sprayable and is not sprayable

virtual bool isSprayable() const = 0;

};

class OilSlick : public GhostRacerActivatedObject

{

public:

// constructor

OilSlick(StudentWorld\* sw, double x, double y);

// specific action when overlaps with ghostracer

virtual void doActivity(GhostRacer\* gr);

// return amount of points given when touches ghost racer

virtual int getScoreIncrease() const;

// return sound when ghostracer overlaps with it

virtual int getSound() const;

// Return whether the object dies after activation.

virtual bool selfDestructs() const;

// Return whether the object is affected by a holy water projectile.

virtual bool isSprayable() const;

};

class HealingGoodie : public GhostRacerActivatedObject

{

public:

// constructor

HealingGoodie(StudentWorld\* sw, double x, double y);

// specific action when overlaps with ghostracer

virtual void doActivity(GhostRacer\* gr);

// return amount of points given when touches ghost racer

virtual int getScoreIncrease() const;

// Return whether the object is affected by a holy water projectile.

virtual bool isSprayable() const;

};

class HolyWaterGoodie : public GhostRacerActivatedObject

{

public:

// constructor

HolyWaterGoodie(StudentWorld\* sw, double x, double y);

// specific action when overlaps with ghostracer

virtual void doActivity(GhostRacer\* gr);

// return amount of points given when touches ghost racer

virtual int getScoreIncrease() const;

// Return whether the object is affected by a holy water projectile.

virtual bool isSprayable() const;

};

class SoulGoodie : public GhostRacerActivatedObject

{

public:

// constructor

SoulGoodie(StudentWorld\* sw, double x, double y);

// specific action when overlaps with ghostracer

virtual void doSomething();

// specific action when overlaps with ghostracer

virtual void doActivity(GhostRacer\* gr);

// return amount of points given when touches ghost racer

virtual int getScoreIncrease() const;

// return sound when ghostracer overlaps with it

virtual int getSound() const;

// Return whether the object is affected by a holy water projectile.

virtual bool isSprayable() const;

}

**2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the holy water goodie class.” or “My zombie pedestrian doesn’t work correctly yet so I treat it like a human pedestrian right now.”**

As far as I am aware, I believe that my game has all the functionality of the finished game. (I might have missed something, but hopefully not!)

**3. A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”**

For many of the Goodies, I assumed that initializing them on the road meant in the range [ROAD\_CENTER - ROAD\_WIDTH / 2, ROAD\_CENTER + ROAD\_WIDTH / 2]. I assumed that the default sound for a goodie being collected is SOUND\_GOT\_GOODIE. I also assumed that we were allowed to modify the constructor for Student World, since I would get a pointer error if cleanup was called before init, which you can do when you press q on the first screen.