Kuoch, Charlie

UCLA / CS32

Prof. Smallberg

Project 3 Report:

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.**

***//Desc:* Will indicate the descriptions of each function**

**//Base Classes are bolded**

**//Destructors are not included and Constructors are not described**

**//Inherited virtual means that the base function is virtual so the derived implementation does not necessarily have to be declared virtual but is just for consistency.**

**ACTOR.H**

**class Actor : public GraphObject**

{

public:

Actor(int imageID, double x, double y, int direction, double size, int depth, StudentWorld\* world); //Desc: Constructor

virtual void doSomething() = 0; //Desc: As specified, doSomething() was made a pure virtual function in this base class, because every single Actor must doSomething per tick, and each Actor’s doSomething method differs.

virtual std::string description() const = 0; //Desc: I created this function because early on in the spec there was an indication that you may need to differentiate between types of Actors since StudentWorld’s container only has pointers to Actors. Therefore, I made this a pure virtual function because I wanted each Actor to return a shor string that would indicate what specifically it was. This function did not have much use in the final implementation, but I thought it would be nice to include anyways.

bool isAlive() const; //Desc: This should be defined here because every derived actor must in some way indicate if it alive or not. The pattern for all Actors is the same so this should not be virtual

void setDead(); //Desc: Every Actor should be able to indicate that it has died by changing its status to dead. The pattern for all Actors is the same so this should not be virtual

void setSpeed(double vSpeed, double hSpeed); //Desc: Every Actor needs to set its speed. Could have been split into two functions but I decided one was better.

StudentWorld\* getWorld() const; //Desc: Accessor

double gethSpeed() const; //Desc: Accessor

double getvSpeed() const; //Desc: Accessor

virtual bool beSprayedIfAppropriate(); //Desc: If the actor can be sprayed, it will apply the effect. Each actor has a different operation if it is sprayed, so this function is made virtual.

virtual bool isCollisionAvoidanceWorthy() const; //Desc: Every actor can or cannot be collisionAvoidanceWorthy, making it virtual ensures that every Actor indicates whether it is CAW or not.

virtual bool moveRelativeToGhostRacerSpeed();//Desc: Every Actor (other than Ghost Racer) moves relative to the GhostRacer’s Speed

virtual bool isSprayable() const = 0; //Desc: Every Actor is or is not sprayable so this was made pure virtual so that every actor must indicate if it is sprayable.

int getLane(); //Desc: Every Actor is on the road somewhere, this function indicates what lane the actor is in. This function is not virtual because every Actor follows the same rules to determine what lane it is in.

};

class BorderLine : public Actor

{

public:

BorderLine(int ImageID, double x, double y, StudentWorld\* world, std::string type); //Desc: Constructor

virtual void doSomething(); //Desc: As specified in Spec. Inherited Virtual

virtual std::string description() const; //Desc: indicates whether it is a white or yellow borderline. Inherited Virtual

virtual bool isSprayable() const { return false; } //Desc: Indicates BorderLine is not Sprayable. Inherited Virtual

};

**class Agent : public Actor**

{

public:

Agent(int imageID, double x, double y, int direction, double size, int depth, int HP, StudentWorld\* world); //Desc: Constructor

virtual bool isCollisionAvoidanceWorthy() const; //Desc: All agents are collisionAvoidance Worthy, so this function returns true and will be inherited by all Agents so no derived Agent class has to declare isCollisionAvoidanceWorthy().

int health() const; //Desc: Accessor to shared data member of all Agents

void getHP(int hp); //Desc: Heals caller by specified # of health points if it does not exceed 100. The pattern for all Agents is the same so this should not be virtual

bool doDamage(int damage); //Desc: All Actors can take damage and follows the same pattern so it is declared here and is not a virtual function. Deals damage indicated by damage if the health of the Agent is not less than or equal to zero

virtual int soundWhenHurt(); //Desc: All Agents must return a sound to play when it takes damage and some Agents play different sounds so this must be made virtual.

virtual int soundWhenDie(); //Desc: All Agents must return a sound to play when it takes damage and some Agents play different sounds so this must be made virtual.

virtual bool isSprayable() const { return true; } //Desc: Almost all Agents are Sprayable so this is declared here and is not pure virtual. Since most Agents have a common value true for this function, it is helpful to have this implementation inherited by all Actors

};

class GhostRacer : public Agent

{

public:

GhostRacer(StudentWorld\* world);

virtual int soundWhenDie(); //Desc: Returns Sound Specified in Spec. Inherited Virtual.

virtual int soundWhenHurt() ;//Desc: Returns Sound Specified in Spec. Inherited Virtual

virtual void doSomething(); //Desc: As Specified in Spec. Inherited Virtual

virtual std::string description() const; //Desc: Returns “GhostRacer”

int getWater()const; //Desc: Returns Amount of Sprays left. Not virtual because it is specific to Ghostracer

void GR\_move(); //Desc: moves GhostRacer according to spec. Not virtual because it is specific to Ghostracer

virtual bool isSprayable() const { return false; }; //Desc: Ghostracer is not Sprayable so it must reimplement isSprayable. Inherited Virtual

void addWater(int water); //Desc: Increases Holy Water Sprays. Not virtual because it is specific to Ghostracer

void spin(); //Desc: Spins GhostRacer according to Spec. Not virtual because it is specific to Ghostracer

};

**class Pedestrian : public Agent**

{

public:

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

virtual int soundWhenHurt() const; //Desc: Returns Sound Specified in Spec. Inherited Virtual

virtual int soundWhenDie() const; //Desc: Returns Sound Specified in Spec. Inherited Virtual

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

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

void moveAndPossiblyPickPlan(); //Desc: Specific to only pedestrians and follows the same algorithm for all pedestrian type Actors, so this is non-virtual.

};

class HumanPedestrian : public Pedestrian

{

public:

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

virtual std::string description() const; //Desc: Returns “HPED”. Inherited Virtual

virtual void doSomething();//Desc: As Specified in Spec. Inherited Virtual

virtual bool beSprayedIfAppropriate(); //Desc: Applies Effect of being sprayed as specified. Inherited Virtual

};

class ZombiePedestrian : public Pedestrian

{

public:

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

virtual std::string description() const; //Desc: returns “ZPED”. Inherited Virtual

virtual void doSomething();//Desc: As Specified in Spec. Inherited Virtual

virtual bool beSprayedIfAppropriate(); //Desc: Applies Effect of being sprayed as specified. Inherited Virtual

};

class ZombieCab : public Agent

{

public:

ZombieCab(StudentWorld\* world, double x, double y, double speed);

virtual void doSomething(); //Desc: As Specified in Spec. Inherited Virtual

std::string description() const; //Desc: returns “ZC”. Inherited Virtual

virtual bool beSprayedIfAppropriate();//Desc: Applies Effect of being sprayed as specified. Inherited Virtual

virtual int soundWhenHurt() const; //Desc: Returns Sound Specified in Spec. Inherited Virtual

virtual int soundWhenDie() const; //Desc: Returns Sound Specified in Spec. Inherited Virtual

};

**class Activator : public Actor**

{

public:

Activator(StudentWorld\* world, int imageID, double x, double y, double size, int dir);

virtual bool beSprayedIfAppropriate(); //Desc: This is declared here because each Ghost Racer Activated Object can get sprayed and effectively takes action the same way by setting its status to dead. This common function will be inherited by all Activators unless the activator is affected by the spray in a different way, in that case it is necessary to make this function virtual.

virtual void doSomething(); //Desc: All Activated Objects follow the same do something pattern which is to moveRelativetoGhostRacer, then doActivity. Therefore, this doSomething method should not be pure virtual because almost every Activator follows the same pattern for their do Something method.

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

virtual void doActivity(GhostRacer\* gr) = 0; //Desc: This is declared here and made pure virtual because every Activator has an activity that they do and each Activator’s Activity differentiates.

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

virtual int getScoreIncrease() const = 0; //Desc: This is declared here and made pure virtual because every Activator can reward the player a specific amount of points which differs per Activator. This returns how many points can be rewarded

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

virtual int getSound() const; //Desc: This is declared here and made pure virtual because every Activator plays a different sound when it is activated. This returns what sound the game should play

// Return whether the object dies after activation.

virtual bool selfDestructs() const = 0; //Desc: This is declared here and made pure virtual because every Activator potentially will die after activation. This differs between each Activator and indicates whether or not that actor will die after being activated

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

virtual bool isSprayable() const { return false; }//Desc: This is declared here and made virtual because most Activators cannot be affected by sprays. Since some activators can be affected, is sprayable cannot be pure virtual since those activators need the ability to indicate that they can be sprayed.

};

class OilSlick : public Activator

{

public:

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

virtual std::string description() const; //Desc: returns “OS”. Inherited Virtual

virtual void doActivity(GhostRacer\* gr); //Desc: As Specified in Spec. Inherited Virtual

virtual int getScoreIncrease() const; //Desc: Returns 0 because it does not reward any points. Inherited Virtual

virtual int getSound() const; //Desc: Returns Sound Specified in Spec. Inherited Virtual

virtual bool selfDestructs() const; //Desc: Returns false because the oil slick doesn’t disappear after being activated. Inherited Virtual

};

class HealingGoodie : public Activator

{

public:

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

virtual std::string description() const; //Desc: returns “HG”. Inherited Virtual

virtual void doActivity(GhostRacer\* gr); //Desc: As Specified in Spec Inherited Virtual

virtual int getScoreIncrease() const; //Desc: Returns 250 points as specified. Inherited Virtual

virtual bool selfDestructs() const; //Desc: Returns true because Healing Goodies disappear after being activated. Inherited Virtual

virtual bool isSprayable() const; //Desc: Returns true because Healing Goodies are sprayable. Inherited Virtual

};

class HolyWaterGoodie : public Activator

{

public:

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

virtual void doActivity(GhostRacer\* gr); //Desc: As Specified in Spec. Inherited Virtual

virtual std::string description() const { return "HW"; } //Desc: Inherited Virtual

virtual int getScoreIncrease() const { return 50; }//Desc: Returns 50 points as specified. Inherited Virtual

virtual bool selfDestructs() const { return true; }//Desc: Returns true because Holy Water Goodies disappear after being activated. Inherited Virtual

virtual bool isSprayable() const { return true; }//Desc: Returns true because Holy Water Goodies are sprayable. Inherited Virtual

};

class SoulGoodie : public Activator

{

public:

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

std::string description() const { return "SG"; } //Desc: Inherited Virtual

virtual void doActivity(GhostRacer\* gr); //Desc: As Specified in Spec. Inherited Virtual

virtual int getScoreIncrease() const { return 100; }}//Desc: Returns 100 points as specified. Inherited Virtual

virtual int getSound() const { return SOUND\_GOT\_SOUL; } //Desc: Returns Sound Specified in Spec. Inherited Virtual

virtual bool selfDestructs() const { return true; }//Desc: Returns true because Soul Goodies disappear after being activated. Inherited Virtual

};

**STUDENTWORLD.H**

class StudentWorld : public GameWorld

{

public:

StudentWorld(std::string assetPath);

~StudentWorld();//Desc: calls cleanUp();

virtual int init(); //Desc: initializes the student world as defined in the spec

virtual int move();//Desc: Ticks the student world as defined in the spec

virtual void cleanUp(); //Desc: deletes all the dynamically allocated list nodes

GhostRacer\* getGhostRacer(); //Desc: returns a pointer to the worlds GR

void recordSoulSaved(); //Desc: records Souls Saved

bool overlaps(const Actor\* a1, const Actor\* a2) const; //Desc: detects actor overlap as indicated in the spec

Actor\* findCAWabove(ZombieCab\* ZC);

//Desc: Looks for a Collision Avoidance Worthy Actor above passed ZombieCab. If one exists, searches if there is a closer one. Otherwise, return nullptr. Returns the closest CollisionAvoidanceWorthy Actor above the ZombieCab.

Actor\* findCAWbelow(ZombieCab\* ZC);

//Desc: Looks for a Collision Avoidance Worthy Actor below passed ZombieCab. If one exists, searches if there is a closer one. Otherwise, return nullptr. Returns the closest CollisionAvoidanceWorthy Actor below the ZombieCab.

Actor\* Sprayable(Spray\* water);

//Desc: Checks if the passed water object overlaps with sprayable actor, if it does, it will return the object that can be sprayed by the water.

void addActor(Actor\*);

//Desc: pushes back the passed actor onto the STL list. Mainly used to allow other members outside the StudentWorld to access and modify the STL.

};

**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.”**

I was able to implement every thing the project without any known or obvious bugs. Looks like my game performs identical to the Sample program other than the fact that it ticks faster when I run it.

**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.”**

The spec never specified if the bonus should ever stop decrementing past zero, so I decided to make the bonus stop decrementing past 0 since it doesn’t make sense to have negative bonus points unless you want to punish the player for taking too long to complete the level.