Actor.h Public Member Functions Descriptions:

**Actor class functions:**

virtual void doSomething(): This is the function that makes an actor do whatever it does during one tick. I chose to define this function as a pure virtual function in the base Actor class because all actors do something in a tick, and different actors have different actions during the tick.

bool isAlive(): This is the function that returns true if the actor is alive, and false if the actor is dead. It is in the Actor class because the game needs to check whether every actor is dead or alive, and it is non-virtual, because the function is the same for all actors, returning true when they are dead and false when they are not.

virtual bool beSprayedIfAppropriate(): This is the function that sprays the actor and returns true if possible, or returns false and does nothing if the actor can’t be sprayed. I chose to make it in the Actor class because all actors can potentially be affected by a spray, and I made it virtual but not pure-virtual, with the base version of the function in actor set to return false and do nothing, because many actors are non-sprayable, so this makes it so I don’t need to write a second function for them.

int getVerticalSpeed(): This function returns the vertical speed of the actor. It is in the Actor class because all actors have a vertical speed, and it is non-virtual because the function is the same for all actors, merely returning their speed.

void setVerticalSpeed(int newSpeed): This function sets the vertical speed of the actor to newSpeed. It is in the Actor class because all actors can have their vertical speed changed, and it is non-virtual because the function is the same for all actors.

void setDead(): This function makes it so that the actor is dead, which is recorded in a variable, and makes the isAlive() function false. It is in the Actor class because all actors can be set dead, and it is non-virtual because there is no difference in the function for all actors.

bool collisionAvoidanceWorthy(): This function returns true if the actor is collision avoidance worthy, and false if it is not. It is in the Actor class because all actors are either collision avoidance worthy or not, and it is non-virtual because the function is the same for all actors, returning a boolean value representing the actor’s property.

bool checkOutsideOfScreen(): This function checks whether or not the actor is outside the screen. If it is outside the screen, it returns true and sets the actor to dead. If it is not, it does nothing and returns false. It is in the Actor class because all actors need to check whether they are in the screen or not, and it is non-virtual because each actor is checked in the same way.

void moveRelativeToGhostRacer(): This function moves actors relative to the ghost racer, based on their current vertical and horizontal speed. It also checks to see if the new move has made the actor outside of the screen, and if it has, it sets the actor to dead. It is non-virtual because it works the same way for all actors, and it is in the actor class because nearly all actors use this function to move.

int getHorizSpeed(): returns the actor’s horizontal speed, in Actor class because nearly all actors have a horizontal speed, and non-virtual because has same function for all actors.

void setHorizSpeed(int s): sets the actor’s horizontal speed to s, in Actor class because all actors change their horizontal speed, non-virtual because same for all actors

double convertToRadians(int degree): Converts the int degree number to radians, by multiplying by pi/180, in Actor class so that all actors can call it, non-virtual because works the same independent of what the actor is

StudentWorld\* getWorld(): returns pointer to the StudentWorld the actor is in, in Actor class because all actors use it, non-virtual because does same thing for all actors.

**Agent class functions:**

int getHP(): returns the HP of the agent, in Agent class because all agents have HP and only agents have this property, and non-virtual because function does the same thing for all agents

void increaseHP(int hp): Increases the agent’s HP by hp parameter, in Agent class because needs to access the hp variable, non-virtual because used same way for all agents

virtual bool takeDamageAndPossiblyDie(int hp): Reduces the agent’s HP by hp parameter, if it has hp lower than or equal to zero set it dead, play death sound, and return true. If not, return false. It is in the Agent class because all agents can take damage, and it is virtual because different agents take damage in different ways, for example, when a zombie pedestrian gets hurt it makes a sound, while the ghost racer doesn’t necessarily make a sound when it gets hurt.

virtual int soundWhenHurt(): returns the sound of the agent when it gets hurt, in agent class because all agents have a sound when hurt, pure virtual because almost each agent has a different sound

virtual int soundWhenDie(): returns the sound of the agent when it dies, in agent class because all agents have a sound when it dies pure virtual because almost each agent has a different sound

**GhostRacer class functions:**

void moveAlgorithm(): moves the ghost racer for a tick as specified by the spec, based on its direction, in GhostRacer class because only the ghost racer moves in this particular way, not virtual as no derived classes

void increaseSprays(int amt): increases the number of sprays the ghost racer has by amt parameter, in GhostRacer class because only ghost racer owns sprays, not virtual because no derived classes

int numSprays(): returns number of sprays the ghost racer currently owns, in GhostRacer class because only ghost racer owns, not virtual because no derived classes

void spin(): spins the car as specified by the spec when the car hits oil, changing its direction slightly. In GhostRacer class because only ghost racer spins, not virtual

**GhostRacerActivatedObject class functions:**

virtual void doSpecialActivity(): does the object’s special activity when hit by ghost racer, such as healing the ghost racer and then setting itself to dead for the healing goodie, or making the car spin if it is the oil slick. In this class because all activated objects have a special activity, and pure-virtual because each type has a different activity.

virtual int getSound(): returns the sound the object makes when the ghost racer touches it, in this class because all activated objects make a sound, pure virtual because nearly all objects make a different sound.

virtual int getScore(): returns score the player gets for touching the object, in this class because nearly all objects give a score to the player when picked up, pure virtual as different score value for each object

StudentWorld.h Public Member Functions Descriptions:

virtual int init(): initialized the screen, at start of game or new life, putting ghost racer and border lines in correct locations, virtual because derived from GameWorld class

virtual int move(): does everything required by the spec in one tick, such as moving every actor, removing dead ones, etc, virtual because derived from GameWorld class

virtual void cleanUp(): cleans up the dynamically allocated data after the player dies, deleting all actors and the list of actors, virtual because derived from GameWorld class

GhostRacer\* getGhostRacer(): returns a pointer to the ghost racer, in StudentWorld class because this class has access to the ghost racer, which the Actor class needs to access sometimes, not virtual because no derived classes

void addActor(Actor\* a): adds Actor a to the world’s list of actors, in StudentWorld class because only this class should access the list of actors, not virtual

bool sprayFirstAppropriateActor(Actor\* a): sprays the first sprayable actor that overlaps with the spray a in the parameter. If successful, returns true, else returns false. In StudentWorld clas because only this class should access the list of all actors, not virtual function

bool overlaps(const Actor\* a1, const Actor\* a2): checks if actor a1 points to and a2 points to overlap based on their x, and y, as specified by the spec, and returns true if they do, false if they don’t. In StudentWorld class so that all actors can use it easily, not virtual function

Actor\* getClosestCollisionToCab(bool inFront, int cabLane, double y):

This function finds a pointer to the actor that is either closest in front, or closest behind in the specified lane and y value in the parameter. If it fails to find one, it returns the null pointer.

Pseudocode:

Set closest actor pointer to null pointer, closest distance to view\_height

For each current actor in the world:

If the actor is alive, and collision avoidance worthy, and in the same lane as the lane in

the parameter, and in front if inFront is true or behind if inFront is false, then:

If its distance from y is closer than the current closestDistance:

Set the closest distance to this distance, and set the closest actor pointer

to point to this actor

return closest actor pointer

This function is in the StudentWorld class as it needs to access the list of all actors in the world, and it is not virtual.

Int getLane(double x): Returns the lane that the x value is in (lane 1 is left lane, lane 2 is middle, lane 3 is right), in StudentWorld class as it allows for easy access for actors, not virtual function

void soulSaved(): Reduces the number of souls left to pass level by one, in StudentWorld class as it is the one keep track of the number of souls, not virtual function

I finished all functionality as far as I know.

Design decisions and assumptions I made:

My Actor class had a similar design as the posted one on the cs32 website, which was as follows:

GraphObject

Actor

BorderLine

Agent

GhostRacer

Pedestrian

HumanPedestrian

ZombiePedestrian

ZombieCab

Spray

GhostRacerActivatedObject

OilSlick

HealingGoodie

HolyWaterGoodie

SoulGoodie

The ghost racer, human pedestrian, zombie pedestrian, and zombie cab were all under the agent class, because they shared similar properties and actions, which included having health points, taking damage and possibly dying, making sounds when getting hurt and dying, and all except the ghost racer could get sprayed by water.

The Human Pedestrian and Zombie Pedestrian were under the Pedestrian class because of their common functionalities, for example, they had a lot of common actions in their doSomething() function, which included moving horizontally, and coming up with a new movement plan when required.

The OilSlick, HealingGoodie, and HolyWaterGoodie, and SoulGoodie were all under the GhostRacerActivatedObject because they shared similar behaviors, as in a tick, each of these objects essentially move down the road, and then if they overlap with the ghost racer, they do their special activity such as making the ghost racer spin, and also make their specified sound, and potentially increase the score of the racer.

Certain actors did not have a horizontal speed or vertical speed; however, I still include these two variable in the base Actor class, because nearly all actors had these two variables, and for the ones that didn’t have a vertical/horizontal speed, I just never accessed this variable or changed it. I also believe the spec did not specify what to do if the bonus points went below zero, but I assumed that if this happened, the player would just get 0 bonus points as opposed to a negative value. I believe the project also didn’t specify whether the cos() and sin() functions were in radians or degrees, but based on the values I obtained I found they were in radians so I converted my direction in degrees to radians to use these two functions properly. When a human pedestrian is hit, I made it play the SOUND\_NONE, as I thought it would be better to play nothing and wait for the program to play SOUND\_PLAYER\_DIE.