Ethan Duong

805124044

Project 3 Report

**High Level Overview of Public Member Functions:**

Class StudentWorld: public GameWorld; Stores pointers to actors in a stl list.

* StudentWorld(std::string assetPath)
  + The predefined constructor in project skeleton
* ~StudentWorld()
  + Calls cleanUp()
* cleanUp()
  + Deletes GhostRacer pointer and frees memory from actor list.
  + Clears actor list of pointers after freeing memory
* init()
  + Adds BorderLines to actors list
  + Sets bonus to max
  + Creates the GhostRacer
* move()
  + First, loops through actor list, telling each actor to do something. After an actor does something, checks to see if player is dead or the level is finished (returns GWSTATUS\_PLAYER\_DIED or GWSTATUS\_FINISHED\_LEVEL respectively )
  + Next, tells GhostRacer to do something
  + Next, loops through actor list, removing dead actors
  + Next, adds new actors
  + Last, output score to screen, decrease bonus by 1, and return GWSTATUS\_CONTINUE\_GAME
* bool cabVicinityCheck(ZombieCab\* a, int lane, bool front)
  + If front is true, then return true if there is an avoidance worthy actor in ZombieCab a’s lane within a certain distance in front of the cab. Otherwise return false
  + If front is false, then return true if there is an avoidance worthy actor in ZombieCab a’s lane within a certain distance behind the cab. Otherwise return false
* bool projectileSearch(Projectile\* proj)
  + Search actor list to see if there is an actor that satisfies the following conditions:
    - Condition 1: the actor can be hurt by a projectile
    - Condition 2: the actor overlaps with Projectile proj.
  + For the first actor where both conditions are fulfilled, damage it and return true
  + Otherwise return false
* void addActor(Actor\* actor)
  + Push Actor\* actor to back of actors list

Class Actor : public GraphObject

* Actor(int imageID, double startX, double startY, int dir = 0, double size = 1.0, unsigned int depth = 0, StudentWorld\* world = nullptr)
  + Constructor; Set private members to correct values
* virtual void doSomething() = 0
  + Makes Actor class an ABC
  + All subclasses have a unique doSomething function. Therefore, it makes sense to define doSomething() as pure virtual in the base class. (and use polymorphism)
* void kill()
  + Sets bool alive to false
* bool isAlive()
  + Return true if alive
* double getVert()
  + Return vertical speed
* double getHor()
  + Return horizontal speed
* void setSpeed(double v, double h)
  + Ret vertical speed to v and horizontal speed to h
* bool isOffScreen()
  + Returns true if x,y coordinates are not within the screen
* void move()
  + Adjusts x,y position based on own vertical and horizontal speed as well as the speed of the GhostRacer
  + GhostRacer used is determined by the StudentWorld assigned to this Actor
    - Each actor keeps a pointer to a StudentWorld
* virtual bool collidesWithProjectile()
  + return false by default
  + We declare this as a virtual function to allow subclasses to redefine it to return true (if we decide that it should be able to collide with a projectile) and make use of polymorphism
* virtual bool receiveDamage(int damage)
  + Do nothing by default
  + We declare this as a virtual function since certain classes receive damage and react in different ways (and to use polymorphism)
  + We do not declare this function as pure virtual given that many subclasses do not take damage and/or are unaffected by “damaging effects” (thus it makes sense to have this function do nothing by default)
* virtual bool isCollisionAvoidanceWorthy()
  + Return false by default
  + We declare this as a virtual function to allow subclasses to redefine it to return true (and use polymorphism)
* bool doesOverlap(Actor\* other)
  + Return true if this actor and the other actor overlap
  + Return false otherwise
* StudentWorld\* getWorld()
  + return a pointer to this actor’s assigned StudentWorld

Class Projectile: public Actor

* Projectile(double startX, double startY, int dir, StudentWorld\* world)
  + Constructor; Set private members to correct values
* virtual void doSomething()
  + If Projectile is dead, return
  + Call projectileSearch() on assigned StudentWorld()
  + moveForward(SPRITE\_HEIGHT)
  + if Projectile is offscreen or has travelled its maximum distance
    - kill()

Class RoadObject : public Actor

* RoadObject(int imageID, double startX, double startY, int dir = 0, double size = 1.0, unsigned int depth = 0, StudentWorld\* world = nullptr)
  + Constructor; Set private members to correct values
* virtual void doSomething()
  + If the RoadObject is not alive, return
  + Next, call actor class’ move() function
  + If the RoadObject died while moving, immediately return
  + If the RoadObject overlaps with the racer
    - attemptCollision()
      * This private virtual function by default does nothing
      * Subclasses can define attemptCollision() on their own (e.g. the healing goodie will heal the racer when attemptCollision() is called)
  + Lastly, call rotate()
    - This private virtual function by default does nothing
    - Only the soulgoodie subclass will define rotate (changes direction by a certain amount)

Class DestroyableGoodie : public RoadObject

* DestroyableGoodie(int imageID, double startX, double startY, int dir = 0, double size = 1.0, unsigned int depth = 0, StudentWorld\* world = nullptr)
  + Constructor; Set private members to correct values
* virtual void receiveDamage(int)
  + Call actor’s kill() function
* Virtual bool collidesWithProjectile()
  + return true

RoadObject immediate subclasses are SoulGoodie, BorderLine, and OilSlick. They define no public members other than their constructors.

DestroyableGoodie immediate subclasses are HealingGoodie and HolyWaterGoodie. They define no public members other than their constructors.

Class Body : public Actor

* Body(int imageID, double startX, double startY, int dir = 0, double size = 1.0, unsigned int depth = 0, StudentWorld\* world = nullptr)
  + Constructor; Set private members to correct values
* virtual isCollisionAvoidanceWorthy()
  + return true
* int getHP()
  + return HP
* void setHP(int health)
  + Set HP = health
* virtual void receiveDamage(int damage)
  + HP -= damage
  + damageEffect()
    - sublclasses define what damageEffect does
    - e.g. for GhostRacer, damageEffect() tests to see if the GhostRacer is out of HP. If so, kill the racer and play a death sound

Class GhostRacer : public Body

* GhostRacer(StudentWorld\* world)
  + Set private members
* Virtual void doSomething()
  + // Follows the spec’s outline
  + if not alive, then return
  + if moving out of bounds
    - receive damage, change direction, and play crash sound
  + if we have valid player input
    - do player input (such as firing a projectile or moving left/right)
  + change racer’s position based on direction
* The rest of the public members are setters and getters for GhostRacer’s holyWater and soulsSaved inventory

Class Planners : public Body

* Planners(int imageID, double startX, double startY, int dir = 0, double size = 1.0, unsigned int depth = 0, StudentWorld\* world = nullptr)
  + Constructor
* virtual void doSomething()
  + if Planner is dead
    - return
  + call preMove() function
  + call actor’s Move() function
  + call postMove() function
  + pickNewPlan()
  + NOTE: preMove(), postMove(), and pickNewPlan() are private virtual functions that by default do nothing but can be redefined by subclasses.
  + E.g. In the preMove() function, a zombie pedestrian tests if there is a GhostRacer in front of it and sets its speed accordingly before the Move() function is called.
* Virtual bool collidesWithProjectile()
  + return true
* Last public member is a setter and getter for movementPlan value

HumanPedestrian, ZombiePedestrian, and ZombieCab are all Planners.

Only HumanPedestrian defines a public member function (other than a constructor); Most of them define private virtuals preMove(), postMove(), etc.

Class HumanPedestrian : public Planners

* virtual void receiveDamage(int damage)
  + // This function gets called when a projectile hits the pedestrian
  + Swap horizontal direction (left to right or right to left)
  + Play the pedestrian hurt sound

Design Assumptions:

1. In regards to how the racer is spun by oil slicks, I decided to make the min/max direction the racer can be spun to be 66 and 114. This lines up with how the racer is spun in the sample game.